Python 2.6 Quick Reference

Contents

Front matter

Version 2.6 (What's new?)
Check updates at http://rgruet.free.fr/#QuickRef.
Please **report** errors, inaccuracies and suggestions to Richard Gruet (pqr at rgruet.net).

Last updated on February 10, 2009.

Feb 10, 2008
: upgraded by Richard Gruet and Josh Stone for Python 2.6

Dec 14, 2006
: upgraded by Richard Gruet for Python 2.5

Feb 17, 2005,
: upgraded by Richard Gruet for Python 2.4

Oct 3, 2003
: upgraded by Richard Gruet for Python 2.3

May 11, 2003, rev 4
: upgraded by Richard Gruet for Python 2.2 (restyled by Andrei)

Aug 7, 2001
: upgraded by Simon Brunning for Python 2.1

May 16, 2001
: upgraded by Richard Gruet and Simon Brunning for Python 2.0

Jun 18, 2000
: upgraded by Richard Gruet for Python 1.5.2

Oct 20, 1995
: created by Chris Hoffmann for Python 1.3

Color coding:
: Features added in 2.6 since 2.5
: Features added in 2.5 since 2.4
: Features added in 2.4 since 2.3

Originally based on:
- Python Bestiary, author: Ken Manheimer

- Python manuals, authors: Guido van Rossum and Fred Drake
- python-mode.el, author: Tim Peters
- and the readers of comp.lang.python

Useful links :

- **Python's nest**: http://www.python.org
- **Official documentation**: http://docs.python.org/2.6/
- **Other doc & free books**: FAQs, Faqts, Dive into Python, Python Cookbook, Thinking in Python, Text processing in Python
- **Getting started**: Python Tutorial, 7mn to Hello World (windows)
- **Topics**: HOWTOs, Databases, Web programming, XML, Web Services, Parsers, Numeric & Scientific Computing, GUI programming, Distributing
- **Where to find packages**: Python Package Index (PyPI), Python Eggs, SourceForge (search "python"), Easy Install, O'Reilly Python DevCenter
- **Wiki**: moinmoin
- **Newsgroups**: comp.lang.python and comp.lang.python.announce
- **Misc pages**: Daily Python URL
- **Python Development**: http://www.python.org/dev/
- **Jython** - Java implementation of Python: http://www.jython.org/
- **IronPython** - Python on .Net: http://www.codeplex.com/Wiki/View.aspx?ProjectName=IronPython
- **ActivePython**: http://www.ActiveState.com/ASPN/Python/
- **Help desk**: help@python.org
- 2 excellent (but somehow outdated) **Python reference books**: Python Essential Reference (Python 2.1) by David Beazley & Guido Van Rossum (Other New Riders) and Python in a nutshell by Alex martelli (O'Reilly).
- **Python 2.4 Reference Card (cheatsheet)** by Laurent Pointal, designed for printing (15 pages).
- Online Python 2.2 Quick Reference by the New Mexico Tech Computer Center.

Tip: From within the Python interpreter, type `help`, `help(`*object*`)` or `help("`*name*`")` to get help.

Invocation Options

python[**w**] [-BdEhimOQsStuUvVWxX3?] [-c *command* | *scriptFile* | -] [*args*]
(python**w** does not open a terminal/console; python does)

Invocation Options

Option	Effect
-B	Prevents module imports from creating `.pyc` or `.pyo` files (see also envt variable `PYTHONDONTWRITEBYTECODE=x` and attribute `sys.dont_write_bytecode`).
-d	Output parser debugging information (also PYTHONDEBUG=x)
-E	Ignore environment variables (such as PYTHONPATH)
-h	Print a help message and exit (formerly -?)
-i	Inspect interactively after running script (also PYTHONINSPECT=x) and force prompts, even if stdin appears not to be a terminal.
-m *module*	Search for *module* on `sys.path` and runs the module as a script. (Implementation improved in 2.5: module `runpy`)
-O	Optimize generated bytecode (also PYTHONOPTIMIZE=x). Asserts are suppressed.
-OO	Remove doc-strings in addition to the -O optimizations.
-Q *arg*	Division options: -Qold (default), -Qwarn, -Qwarnall, -Qnew
-s	Disables the user-specific module path (also PYTHONNOUSERSITE=x)
-S	Don't perform `import site` on initialization.
-t	Issue warnings about inconsistent tab usage (-tt: issue errors).
-u	Unbuffered binary stdout and stderr (also PYTHONUNBUFFERED=x).
-U	Force Python to interpret all string literals as Unicode literals.
-v	Verbose (trace import statements) (also PYTHONVERBOSE=x).
-V	Print the Python version number and exit.
-W *arg*	Warning control (arg is action:message:category:module:lineno)
-x	Skip first line of source, allowing use of non-unix Forms of `#!cmd`
~~-X~~	~~Disable class based built-in exceptions (for backward compatibility management of exceptions)~~
-3	Emit a `DeprecationWarning` for Python 3.x incompatibilities
-c	Specify the command to execute (see next section). This terminates the option list (following

command	options are passed as arguments to the command).
scriptFile	The name of a python file (.py) to execute. Read from stdin.
-	Program read from stdin (default; interactive mode if a tty).
args	Passed to script or command (in `sys.argv[1:]`)
	If no scriptFile or command, Python enters interactive mode.

- Available **IDEs** in std distrib: **IDLE** (tkinter based, portable), **Pythonwin** (on Windows). Other free IDEs: IPython (enhanced interactive Python shell), Eric, SPE, BOA constructor, PyDev (Eclipse plugin).
- Typical python **module header** :

```
#!/usr/bin/env python
# -*- coding: latin1 -*-
```

  Since 2.3 the *encoding* of a Python source file must be declared as one of the two first lines (or defaults to **7 bits Ascii**) [PEP-0263], with the format:

```
# -*- coding: encoding -*-
```

  Std *encodings* are defined here, e.g. ISO-8859-1 (aka latin1), iso-8859-15 (latin9), UTF-8... Not all encodings supported, in particular UTF-16 is not supported.
- It's now a **syntax error** if a module contains string literals with 8-bit characters but doesn't have an encoding declaration (was a warning before).
- Since 2.5, `from __future__ import` *`feature`* statements must be declared at **beginning** of source file.
- **Site customization**: File `sitecustomize.py` is automatically loaded by Python if it exists in the Python path (ideally located in `${PYTHONHOME}/lib/site-packages/`).
- **Tip:** when launching a Python script on Windows,

  `<pythonHome>\python myScript.py args ...` can be reduced to :
  `myScript.py args ...` if <pythonHome> is in the PATH envt variable, and further reduced to :
  `myScript args ...` provided that `.py;.pyw;.pyc;.pyo` is added to the `PATHEXT` envt variable.

Environment variables

Environment variables	
Variable	**Effect**
PYTHONHOME	Alternate *prefix* directory (or *prefix*:*exec_prefix*). The default module search path uses *prefix*/lib
PYTHONPATH	Augments the default search path for module files. The format is the same as the shell's `$PATH`: one or more directory pathnames separated by ':' or ';' without spaces around (semi-) colons ! On Windows Python first searches for Registry key `HKEY_LOCAL_MACHINE\Software\Python\PythonCore\`*`x.y`*`\PythonPath` (default value). You can create a key named after your application with a default string value giving the root directory path of your appl. Alternatively, you can create a text file with a `.pth` extension, containing the path(s), one per line, and put the file somewhere in the Python search path (ideally in the `site-packages/` directory). It's better to create a .pth for each application, to make easy to uninstall them.
PYTHONSTARTUP	If this is the name of a readable file, the Python commands in that file are executed before the first prompt is displayed in interactive mode (no default).
PYTHONDEBUG	If non-empty, same as -d option
PYTHONINSPECT	If non-empty, same as -i option
PYTHONOPTIMIZE	If non-empty, same as -O option
PYTHONUNBUFFERED	If non-empty, same as -u option
PYTHONVERBOSE	If non-empty, same as -v option
PYTHONCASEOK	If non-empty, ignore case in file/module names (imports)
PYTHONDONTWRITEBYTECODE	If non-empty, same as -B option
PYTHONIOENCODING	Alternate `encodingname` or `encodingname:errorhandler` for stdin, stdout, and stderr, with the same choices accepted by `str.encode()`.
PYTHONUSERBASE	Provides a private `site-packages` directory for **user-specific** modules. [PEP-0370] - On Unix and Mac OS X, defaults to `~/.local/`, and modules are found in a

	version-specific subdirectory like `lib/python2.6/site-packages`. - On Windows, defaults to `%APPDATA%/Python` and `Python26/site-packages`.
PYTHONNOUSERSITE	If non-empty, same as -s option

Notable lexical entities

Keywords

```
and       del       for       is        raise
assert    elif      from      lambda    return
break     else      global    not       try
class     except    if        or        while
continue  exec      import    pass      with
def       finally   in        print     yield
```

- (List of keywords available in std module: **keyword**)
- Illegitimate Tokens (only valid in strings): `$ ?` (plus `@` before 2.4)
- A statement must all be on a single line. To break a statement over multiple lines, use "****", as with the C preprocessor.
  Exception: can always break when inside any (), [], or {} pair, or in triple-quoted strings.
- More than one statement can appear on a line if they are separated with semicolons ("**;**").
- Comments start with "`#`" and continue to end of line.

Identifiers

(*letter* | "_") (*letter* | *digit* | "_")*

- Python identifiers keywords, attributes, etc. are **case-sensitive**.
- Special forms: *_ident* (not imported by 'from module import *'); **__***ident***__** (system defined name); *__ident* (class-private name mangling).

String literals

Two flavors: `str` (standard 8 bits locale-dependent strings, like ascii, iso 8859-1, utf-8, ...) and `unicode` (16 or 32 bits/char in utf-16 mode or 32 bits/char in utf-32 mode); one common ancestor `basestring`.

Literal
"a string enclosed by double quotes**"**
'another string delimited by single quotes and with a " inside**'**
'''a string containing embedded newlines and quote (') marks, can be delimited with triple quotes.**'''**
""" may also use 3- double quotes as delimiters **"""**
b"An 8-bit string**"** - A `bytes` instance, a forward-compatible form for an 8-bit string**'**
B"Another 8-bit string**"**
u'a unicode string**'**
U"Another unicode string**"**
r'a raw string where \ are kept (literalized): handy for regular expressions and windows paths!**'**
R"another raw string**"** **--** raw strings cannot end with a \
ur'a unicode raw string**'**
UR"another raw unicode**"**

- Use **** at end of line to continue a string on next line.
- Adjacent strings are concatened, e.g. `'Monty ' 'Python'` is the same as `'Monty Python'`.
- `u'hello' + ' world'` --> `u'hello world'` (coerced to unicode)

String Literal Escapes

Escape	**Meaning**
newline	Ignored (escape newline)
****	Backslash (\\)
\\e	Escape (ESC)
\\v	Vertical Tab (VT)

\'	Single quote (')
\f	Formfeed (FF)
*****ooo*	char with octal value *ooo*
\"	Double quote (")
\n	Linefeed (LF)
\a	Bell (BEL)
\r	Carriage Return (CR)
\x*hh*	char with hex value *hh*
\b	Backspace (BS)
\t	Horizontal Tab (TAB)
\u*xxxx*	Character with 16-bit hex value *xxxx* (unicode only)
\U*xxxxxxxx*	Character with 32-bit hex value *xxxxxxxx* (unicode only)
\N{*name***}**	Character named in the Unicode database (unicode only), e.g. `u'\N{Greek Small Letter Pi}'` <=> `u'\u03c0'`. (Conversely, in module unicodedata, `unicodedata.name(u'\u03c0') == 'GREEK SMALL LETTER PI'`)
AnyOtherChar	left as-is, including the backslash, e.g. `str('\z') == '\\z'`

- NUL byte (`\000`) is **not** an end-of-string marker; NULs may be embedded in strings.
- Strings (and tuples) are immutable: they cannot be modified.

Boolean constants (since 2.2.1)

- **True**
- **False**

In 2.2.1, True and False are integers 1 and 0. Since 2.3, they are of new type `bool`.

Numbers

- **Decimal integer**: `1234, 1234567890546378940L` (or `l`)
- **Binary** integer: `0b10, 0B10, 0b101010101010101010101010101010L` (begins with a `0b` or `0B`)
- **Octal** integer: `0177, 0o177, 0O177, 0177777777777777777L` (begins with a `0` , `0o`, or `0O`)
- **Hex** integer: `0xFF, 0XFFFFffffFFFFFFFFFFL` (begins with `0x` or `0X`)
- **Long** integer (unlimited precision): `1234567890123456L` (ends with `L` or `l`) or `long(1234)`
- **Float** (double precision): `3.14e-10, .001, 10., 1E3`
- **Complex**: `1J, 2+3J, 4+5j` (ends with `J` or `j`, `+` separates (float) real and imaginary parts)

Integers and long integers are **unified** starting from release 2.2 (the `L` suffix is no longer required)

Sequences

- **Strings** (types `str` and `unicode`) of length 0, 1, 2 (see above)
  '', '1', "12", 'hello\n'
- **Tuples** (type `tuple`) of length 0, 1, 2, etc:
  () (1,) (1,2) # parentheses are optional if len > 0
- **Lists** (type `list`) of length 0, 1, 2, etc:
  [] [1] [1,2]

- Indexing is **0**-based. Negative indices (usually) mean count backwards from end of sequence.
- Sequence **slicing [***starting-at-index* **:** *but-less-than-index [: step]***]**. Start defaults to 0, end to len(sequence), step to 1.

```
a = (0,1,2,3,4,5,6,7)
a[3] == 3
a[-1] == 7
a[2:4] == (2, 3)
a[1:] == (1, 2, 3, 4, 5, 6, 7)
a[:3] == (0, 1, 2)
a[:] == (0,1,2,3,4,5,6,7) # makes a copy of the sequence.
a[::2] == (0, 2, 4, 6) # Only even numbers.
a[::-1] = (7, 6, 5, 4, 3 , 2, 1, 0) # Reverse order.
```

Dictionaries (Mappings)

Dictionaries (type `dict`) of length 0, 1, 2, etc: **{}** {1 : 'first'} {1 : 'first', 'two': 2, *key*:*value*}

Keys must be of a *hashable* type; Values can be any type.

Operators and their evaluation order

<table>
<caption>Operators and their evaluation order</caption>
<tr><th>Highest</th><th>Operator</th><th>Comment</th></tr>
<tr><td></td><td>, [...] {...} `...`</td><td>Tuple, list & dict. creation; string conv.</td></tr>
<tr><td></td><td>s[i] s[i:j] s.attr f(...)</td><td>indexing & slicing; attributes, fct calls</td></tr>
<tr><td></td><td>+x, -x, ~x</td><td>Unary operators</td></tr>
<tr><td></td><td>x**y</td><td>Power</td></tr>
<tr><td></td><td>x*y x/y x%y</td><td>mult, division, modulo</td></tr>
<tr><td></td><td>x+y x-y</td><td>addition, substraction</td></tr>
<tr><td></td><td>x<<y x>>y</td><td>Bit shifting</td></tr>
<tr><td></td><td>x&y</td><td>Bitwise and</td></tr>
<tr><td></td><td>x^y</td><td>Bitwise exclusive or</td></tr>
<tr><td></td><td>x|y</td><td>Bitwise or</td></tr>
<tr><td></td><td>x<y x<=y x>y x>=y x==y x!=y x<>y
x is y x is not y
x in s x not in s</td><td>Comparison,
identity,
membership</td></tr>
<tr><td></td><td>not x</td><td>boolean negation</td></tr>
<tr><td></td><td>x and y</td><td>boolean and</td></tr>
<tr><td></td><td>x or y</td><td>boolean or</td></tr>
<tr><td>Lowest</td><td>lambda args: expr</td><td>anonymous function</td></tr>
</table>

- Alternate names are defined in module operator (e.g. `__add__` and `add` for +)
- Most operators are overridable

Basic types and their operations

Comparisons (defined between *any* types)

Comparisons

Comparison	Meaning	Notes
<	strictly less than	(1)
<=	less than or equal to	
>	strictly greater than	
>=	greater than or equal to	
==	equal to	
!= or **<>**	not equal to	
is	object identity	(2)
is not	negated object identity	(2)

Notes:
- Comparison behavior can be overridden for a given class by defining special method __cmp__.
- (1) X < Y < Z < W has expected meaning, unlike C
- (2) Compare object identities (i.e. **id**(object)), not object values.

None

- `None` is used as default return value on functions. Built-in single object with type `NoneType`. Might become a keyword in the future.
- Input that evaluates to `None` does not print when running Python interactively.
- None is now a **constant**; trying to bind a value to the name "None" is now a syntax error.

Boolean operators

Boolean values and operators

Value or Operator	Evaluates to	Notes
built-in **bool**(*expr*)	**True** if *expr* is true, **False** otherwise.	see True, False
None, numeric zeros, empty sequences and mappings	considered False	
all other values	considered True	
not *x*	**True** if *x* is **False**, else **False**	
x **or** *y*	if *x* is **False** then *y*, else *x*	(1)
x **and** *y*	if *x* is **False** then *x*, else *y*	(1)

Notes:
- Truth testing behavior can be overridden for a given class by defining special method __nonzero__.
- (1) Evaluate second arg only if necessary to determine outcome.

Numeric types

Floats, integers, long integers, Decimals.

- Floats (type `float`) are implemented with C doubles.
- Integers (type `int`) are implemented with C longs (signed 32 bits, maximum value is `sys.maxint`)
- Long integers (type `long`) have unlimited size (only limit is system resources).
- Integers and long integers are **unified** starting from release 2.2 (the **L** suffix is no longer required). `int()` returns a `long` integer instead of raising `OverflowError`. Overflowing operations such as `2<<32` no longer trigger `FutureWarning` and return a long integer.
- Since 2.4, new type `Decimal` introduced (see module: decimal) to compensate for some limitations of the floating point type, in particular with fractions. Unlike floats, decimal numbers can be represented exactly; exactness is preserved in calculations; precision is user settable via the `Context` type [PEP 327].

Operators on all numeric types

Operators on all numeric types	
Operation	**Result**
abs(*x*)	the absolute value of *x*
int(*x*)	*x* converted to integer
long(*x*)	*x* converted to long integer
float(*x*)	*x* converted to floating point
-*x*	*x* negated
+*x*	*x* unchanged
x + *y*	the sum of *x* and *y*
x - *y*	difference of *x* and *y*
x * *y*	product of *x* and *y*
x / *y*	true division of x by y: 1/2 -> 0.5 (1)
x // *y*	floor division operator: 1//2 -> 0 (1)
x % *y*	*x* modulo *y*
divmod(*x*, *y*)	the tuple (*x*//*y*, *x*%*y*)
x ** *y*	*x* to the power *y* (the same as **pow**(*x*,*y*))

Notes:
- (1) **/** is still a *floor* division (1/2 == 0) unless validated by a `from __future__ import division`.
- classes may override methods `__truediv__` and `__floordiv__` to redefine these operators.

Bit operators on integers and long integers

Bit operators	
Operation	**Result**
~*x*	the bits of *x* inverted
x ^ *y*	bitwise exclusive or of *x* and *y*
x & *y*	bitwise and of *x* and *y*
x | *y*	bitwise or of *x* and *y*
x << *n*	*x* shifted left by *n* bits
x >> *n*	*x* shifted right by *n* bits

Complex Numbers

- Type `complex`, represented as a pair of machine-level double precision floating point numbers.
- The real and imaginary value of a complex number z can be retrieved through the attributes z.real and z.imag.

Numeric exceptions

`TypeError`
    raised on application of arithmetic operation to non-number
`OverflowError`
    numeric bounds exceeded
`ZeroDivisionError`
    raised when zero second argument of div or modulo op

Operations on all sequence types (lists, tuples, strings)

Operations on all sequence types

Operation	Result	Notes
x **in** *s*	`True` if an item of *s* is equal to *x*, else `False`	(3)
x **not in** *s*	`False` if an item of *s* is equal to *x*, else `True`	(3)
s1 **+** *s2*	the concatenation of *s1* and *s2*	
s ***** *n*, *n*******s*	*n* copies of *s* concatenated	
*s***[***i***]**	*i*'th item of *s*, origin 0	(1)
*s***[***i***:** *j***]** *s***[***i***:** *j***:***step***]**	Slice of *s* from *i* (included) to *j*(excluded). Optional *step* value, possibly negative (default: 1).	(1), (2)
s.**count**(*x*)	returns number of *i*'s for which *s*[*i*] == *x*	
s.**index**(*x*[, *start*[, *stop*]])	returns smallest *i* such that *s*[*i*]==*x*. *start* and *stop* limit search to only part of the sequence.	(4)
len(*s*)	Length of *s*	
min(*s*)	Smallest item of *s*	
max(*s*)	Largest item of *s*	
reversed(*s*)	[2.4] Returns an iterator on *s* in reverse order. *s* must be a sequence, not an iterator (use `reversed(list(s))` in this case. [PEP 322]	
sorted(*iterable* [, *cmp*] [, cmp=*cmpFct*] [, key=*keyGetter*] [, reverse=*bool*])	[2.4] works like the new in-place list.sort(), but sorts a **new** list created from the *iterable*.	

Notes:

- (1) if *i* or *j* is negative, the index is relative to the end of the string, ie len(*s*)+*i* or len(*s*)+*j* is substituted. But note that -0 is still 0.
- (2) The slice of *s* from *i* to *j* is defined as the sequence of items with index *k* such that *i*<= *k* < *j*.
  If *i* or *j* is greater than len(*s*), use len(*s*). If *j* is omitted, use len(*s*). If *i* is greater than or equal to *j*, the slice is empty.
- (3) For strings: before 2.3, x must be a single character string; Since 2.3, `x in s` is True if x is a *substring* of s.
- (4) Raises a `ValueError` exception when *x* is not found in *s* (i.e. out of range).

Operations on mutable sequences (type `list`)

Operations on mutable sequences

Operation	Result	Notes
*s***[***i***]** **=***x*	item *i* of *s* is replaced by *x*	
s[*i*:*j* [:*step*]] **=** *t*	slice of *s* from *i* to *j* is replaced by *t*	
del *s*[*i*:*j*[:*step*]]	same as *s*[*i*:*j*] = []	
s.**append**(*x*)	same as *s*[len(*s*) : len(*s*)] = [*x*]	
s.**extend**(*x*)	same as *s*[len(*s*):len(*s*)]= *x*	(5)
s.**count**(*x*)	returns number of *i*'s for which *s*[*i*] == *x*	
s.**index**(*x*[, *start*[, *stop*]])	returns smallest *i* such that *s*[*i*]==*x*. *start* and *stop* limit search to only part of the list.	(1)
s.**insert**(*i*, *x*)	same as *s*[*i*:*i*] = [*x*] if *i*>= 0. *i* == -1 inserts **before** the last element.	
s.**remove**(*x*)	same as del *s*[*s*.index(*x*)]	(1)
s.**pop**(**[***i***]**)	same as x = s[i]; del s[i]; return x	(4)
s.**reverse**()	reverses the items of *s* in place	(3)
s.**sort**(**[***cmp* **]**) *s*.**sort**([cmp=*cmpFct*] [, key=*keyGetter*] [, reverse=*bool*])	sorts the items of *s* in place	(2), (3)

Notes:

- (1) Raises a `ValueError` exception when *x* is not found in *s* (i.e. out of range).
- (2) The sort() method takes an optional argument `cmp` specifying a comparison function takings 2 list items and returning -1, 0, or 1 depending on whether the 1st argument is considered smaller than, equal to, or larger than the 2nd argument. Note that this slows the sorting process down considerably. Since 2.4, the `cmp` argument may be specified as a keyword, and 2 optional keywords args are added: `key` is a fct that takes a list item and returns the key to use in the comparison (**faster** than `cmp`); `reverse`: If True, reverse the sense of the comparison used.
  Since Python 2.3, the sort is guaranteed "stable". This means that two entries with equal keys will be returned in the same order as they were input. For example, you can sort a list of people by name, and then sort the list by age, resulting in a list sorted by age where people with the same age are in name-sorted order.
- (3) The `sort()` and `reverse()` methods **modify** the list **in place** for economy of space when sorting or reversing a large list. They don't return the sorted or reversed list to remind you of this side effect.
- (4) The `pop()` method is not supported by mutable sequence types other than lists. The optional argument i defaults to -1, so that by default the last item is removed and returned.
- (5) Raises a `TypeError` when x is not a list object.

Operations on mappings / dictionaries (type `dict`)

Operations on mappings

Operation	Result	Notes
len(*d*)	The number of items in *d*	
dict() **dict**(***kwargs*) **dict**(*iterable*) **dict**(d)	Creates an empty dictionary. Creates a dictionary init with the keyword args *kwargs*. Creates a dictionary init with (key, value) pairs provided by *iterable*. Creates a dictionary which is a copy of dictionary *d*.	
d.**fromkeys**(*iterable*, *value*=None)	Class method to create a dictionary with keys provided by *iterator*, and all values set to *value*.	
d[*k*]	The item of *d* with key *k*	(1)
d[*k*] = x	Set *d*[*k*] to *x*	
del *d*[*k*]	Removes *d*[*k*] from *d*	(1)
d.**clear**()	Removes all items from *d*	
d.**copy**()	A shallow copy of *d*	
d.**has_key**(*k*) *k* **in** *d*	`True` if *d* has key *k*, else `False`	
d.**items**()	A copy of *d*'s list of (key, item) pairs	(2)
d.**keys**()	A copy of *d*'s list of keys	(2)
d1.**update**(*d2*)	`for k, v in d2.items(): d1[k] = v` Since 2.4, **update**(***kwargs*) and **update**(*iterable*) may also be used.	
d.**values**()	A copy of *d*'s list of values	(2)
d.**get**(*k*, *defaultval*)	The item of *d* with key *k*	(3)
d.**setdefault**(*k*[,*defaultval*])	*d*[*k*] if *k* in *d*, else *defaultval*(also setting it)	(4)
d.**iteritems**()	Returns an iterator over (key, value) **pairs**.	
d.**iterkeys**()	Returns an iterator over the mapping's **keys**.	
d.**itervalues**()	Returns an iterator over the mapping's **values**.	
d.**pop**(*k*[, *default*])	Removes key *k* and returns the corresponding value. If key is not found, *default* is returned if given, otherwise `KeyError` is raised.	
d.**popitem**()	Removes and returns an arbitrary (key, value) pair from *d*	

Notes:

- `TypeError` is raised if key is not acceptable.
- (1) `KeyError` is raised if key k is not in the map.
- (2) Keys and values are listed in random order.
- (3) Never raises an exception if *k* is not in the map, instead it returns *defaultval*. *defaultval* is optional, when not provided and *k* is not in the map, `None` is returned.
- (4) Never raises an exception if *k* is not in the map, instead returns *defaultVal*, and adds *k* to map with value *defaultVal*. *defaultVal* is optional. When not provided and *k* is not in the map, `None` is returned and added to map.

Operations on strings (types `str` & `unicode`)

These string methods largely (but not completely) supersede the functions available in the string module.
The `str` and `unicode` types share a common base class `basestring`.

Operations on strings

Operation	**Result**	**Notes**
s.**capitalize**()	Returns a copy of *s* with its first character capitalized, and the rest of the characters lowercased.	
s.**center**(*width*[, *fillChar=' '*])	Returns a copy of *s* centered in a string of length *width*, surrounded by the appropriate number of *fillChar* characters.	(1)
s.**count**(*sub*[, *start*[, *end*]])	Returns the number of occurrences of substring *sub* in string *s*.	(2)
s.**decode**([*encoding*[, *errors*]])	Returns a `unicode` string representing the decoded version of `str` *s*, using the given codec (encoding). Useful when reading from a file or a I/O function that handles only `str`. Inverse of `encode`.	(3)
s.**encode**([*encoding*[, *errors*]])	Returns a `str` representing an encoded version of *s*. Mostly used to encode a `unicode` string to a `str` in order to print it or write it to a file (since these I/O functions only accept `str`), e.g. `u'légère'.encode('utf8')`. Also used to encode a `str` to a `str`, e.g. to zip (codec 'zip') or uuencode (codec 'uu') it. Inverse of `decode`.	(3)
s.**endswith**(*suffix* [, *start*[, *end*]])	Returns `True` if *s* ends with the specified *suffix*, otherwise return false. Since 2.5 *suffix* can also be a **tuple** of strings to try.	(2)
s.**expandtabs**([*tabsize*])	Returns a copy of *s* where all tab characters are expanded using spaces.	(4)
s.**find**(*sub* [,*start* [,*end*]])	Returns the lowest index in *s* where substring *sub* is found. Returns **-1** if *sub* is not found.	(2)
s.**format**(**args, *kwargs*)	Returns *s* after replacing numeric and named formatting references found in braces `{}`. (details)	
s.**index**(*sub*[, *start*[, *end*]])	like **find()**, but raises `ValueError` when the substring is not found.	(2)
s.**isalnum**()	Returns `True` if all characters in *s* are alphanumeric, `False` otherwise.	(5)
s.**isalpha**()	Returns `True` if all characters in *s* are alphabetic, `False` otherwise.	(5)
s.**isdigit**()	Returns `True` if all characters in *s* are digit characters, `False` otherwise.	(5)
s.**islower**()	Returns `True` if all characters in *s* are lowercase, `False` otherwise.	(6)
s.**isspace**()	Returns `True` if all characters in *s* are whitespace characters, `False` otherwise.	(5)
s.**istitle**()	Returns `True` if string *s* is a titlecased string, `False` otherwise.	(7)
s.**isupper**()	Returns `True` if all characters in *s* are uppercase, `False` otherwise.	(6)
separator.**join**(*seq*)	Returns a concatenation of the strings in the sequence *seq*, separated by string *separator*, e.g.: `",".join(['A', 'B', 'C']) -> "A,B,C"`	
s.**ljust/rjust/center**(*width*[, *fillChar=' '*])	Returns *s* left/right justified/centered in a string of length *width*.	(1), (8)
s.**lower**()	Returns a copy of *s* converted to lowercase.	
s.**lstrip**([*chars*])	Returns a copy of *s* with leading *chars* (default: blank chars) removed.	
s.**partition**(*separ*)	Searches for the separator *separ* in *s*, and returns a tuple `(head, sep, tail)` containing the part before it, the separator itself, and the part after it. If the separator is not found, returns (*s*, '', '').	
s.**replace**(*old*, *new*[, *maxCount* =-1])	Returns a copy of *s* with the first *maxCount* (-1: unlimited) occurrences of substring *old* replaced by *new*.	(9)
s.**rfind**(*sub*[, *start*[, *end*]])	Returns the highest index in *s* where substring *sub* is found. Returns **-1** if *sub* is not found.	(2)
s.**rindex**(*sub*[, *start*[, *end*]])	like **rfind()**, but raises `ValueError` when the substring is not found.	(2)
s.**rpartition**(*separ*)	Searches for the separator *separ* in *s*, starting at the end of *s*, and returns a tuple `(head, sep, tail)` containing the (left) part before it, the separator itself, and the (right) part after it. If the separator is not found, returns ('', '', *s*).	
s.**rstrip**([*chars*])	Returns a copy of *s* with trailing *chars*(default: blank chars) removed, e.g. `aPath.rstrip('/')` will remove the trailing '/'from aPath **if it exists**	
s.**split**([*separator*[, *maxsplit*]])	Returns a list of the words in *s*, using *separator* as the delimiter string.	(10)

s.**rsplit**(**[** *separator***[**, *maxsplit***]]**)	Same as `split`, but splits from the end of the string.	(10)
s.**splitlines** (**[** *keepends***]**)	Returns a list of the lines in *s*, breaking at line boundaries.	(11)
s.**startswith**(*prefix* **[**, *start***[**, *end***]]**)	Returns `True` if *s* starts with the specified *prefix*, otherwise returns `False`. Negative numbers may be used for *start* and *end*. Since 2.5 *prefix* can also be a **tuple** of strings to try.	(2)
s.**strip**([*chars*])	Returns a copy of *s* with leading and trailing *chars*(default: blank chars) removed.	
s.**swapcase**()	Returns a copy of *s* with uppercase characters converted to lowercase and vice versa.	
s.**title**()	Returns a titlecased copy of *s*, i.e. words start with uppercase characters, all remaining cased characters are lowercase.	
s.**translate**(*table*[, *deletechars*=''])	Returns a copy of *s* mapped through translation table *table*. Characters from *deletechars* are removed from the copy prior to the mapping. Since 2.6 *table* may also be `None` (identity transformation) - useful for using `translate` to **delete** chars only.	(12)
s.**upper**()	Returns a copy of *s* converted to uppercase.	
s.**zfill**(*width*)	Returns the numeric string left filled with zeros in a string of length *width*.	

Notes:

- (1) Padding is done using spaces or the given character.
- (2) If optional argument *start* is supplied, substring *s***[***start***:]** is processed. If optional arguments *start* and *end* are supplied, substring *s***[***start***:***end***]** is processed.
- (3) Default encoding is `sys.getdefaultencoding()`, can be changed via `sys.setdefaultencoding()`. Optional argument *errors* may be given to set a different error handling scheme. The default for *errors* is **'strict'**, meaning that encoding errors raise a **ValueError**. Other possible values are **'ignore'** and **'replace'**. See also module codecs.
- (4) If optional argument *tabsize* is not given, a tab size of 8 characters is assumed.
- (5) Returns `False` if string *s* does not contain at least one character.
- (6) Returns `False` if string *s* does not contain at least one cased character.
- (7) A titlecased string is a string in which uppercase characters may only follow uncased characters and lowercase characters only cased ones.
- (8) *s* is returned if *width* is less than **len(***s***)**.
- (9) If the optional argument *maxCount* is given, only the first *maxCount* occurrences are replaced.
- (10) If *separator* is not specified or `None`, any whitespace string is a separator. If *maxsplit* is given, at most *maxsplit* splits are done.
- (11) Line breaks are not included in the resulting list unless *keepends* is given and true.
- (12) *table* must be a string of length 256.

String formatting with the % operator

formatString **%** *args* --> evaluates to a string

- *formatString* mixes normal text with C printf *format fields* :

  %[*flag*][*width*][*.precision*] *formatCode*

  where *formatCode* is one of c, s, i, d, u, o, x, X, e, E, f, g, G, r, % (see table below).
- The *flag* characters -, +, blank, # and 0 are understood (see table below).
- *Width* and *precision* may be a ***** to specify that an integer argument gives the actual width or precision. Examples of *width* and *precision* :

Examples

Format string	Result
`'%3d' % 2`	`' 2'`
`'%*d' % (3, 2)`	`' 2'`
`'%-3d' % 2`	`'2 '`
`'%03d' % 2`	`'002'`
`'% d' % 2`	`' 2'`
`'%+d' % 2`	`'+2'`
`'%+3d' % -2`	`' -2'`
`'%- 5d' % 2`	`' 2 '`
`'%.4f' % 2`	`'2.0000'`
`'%.*f' % (4, 2)`	`'2.0000'`
`'%0*.*f' % (10, 4, 2)`	`'00002.0000'`
`'%10.4f' % 2`	`' 2.0000'`
`'%010.4f' % 2`	`'00002.0000'`

- %s will convert any type argument to string (uses *str()* function)
- *args* may be a single arg or a tuple of args

```
'%s has %03d quote types.' % ('Python', 2)  == 'Python has 002 quote types.'
```

- Right-hand-side can also be a *mapping*:

```
a = '%(lang)s has %(c)03d quote types.' % {'c':2, 'lang':'Python'}
```

(`vars()` function very handy to use on right-hand-side)

Format codes

Code	Meaning
d	Signed integer decimal.
i	Signed integer decimal.
o	Unsigned octal.
u	Unsigned decimal.
x	Unsigned hexadecimal (lowercase).
X	Unsigned hexadecimal (uppercase).
e	Floating point exponential format (lowercase).
E	Floating point exponential format (uppercase).
f	Floating point decimal format.
F	Floating point decimal format.
g	Same as "e" if exponent is greater than -4 or less than precision, "f" otherwise.
G	Same as "E" if exponent is greater than -4 or less than precision, "F" otherwise.
c	Single character (accepts integer or single character string).
r	String (converts any python object using `repr()`).
s	String (converts any python object using `str()`).
%	No argument is converted, results in a "%" character in the result. (The complete specification is %%.)

Conversion flag characters

Flag	Meaning
#	The value conversion will use the "alternate form".
0	The conversion will be zero padded.
-	The converted value is left adjusted (overrides "-").
	(a space) A blank should be left before a positive number (or empty string) produced by a signed conversion.
+	A sign character ("+" or "-") will precede the conversion (overrides a "space" flag).

String templating

Since 2.4 [PEP 292] the string module provides a new mechanism to substitute variables into *template* strings. Variables to be substituted begin with a **$**. Actual values are provided in a dictionary via the `substitute` or `safe_substitute` methods (`substitute` throws `KeyError` if a key is missing while `safe_substitute` ignores it) :

```
t = string.Template('Hello $name, you won $$$amount') # (note $$ to literalize $)
t.substitute({'name': 'Eric', 'amount': 100000}) # -> u'Hello Eric, you won $100000'
```

String formatting with format()

Since 2.6 [PEP 3101] string formatting can also be done with the `format()` method:

"string-to-format".format(*args*)

Format fields are specified in the string, surrounded by `{}`, while actual values are args to `format()`:

{*field*[!*conversion*][:*format_spec*]}

- Each *field* refers to an arg either by its position (>=0), or by its name if it's a *keyword* argument. The same arg can be referenced more than once.
- The *conversion* can be `!s` or `!r` to call `str()` or `repr()` on the field before formatting.
- The *format_spec* takes the following form:

  [[*fill*]*align*][*sign*][#][0][*width*][.*precision*][*type*]

  - The *align* flag controls the alignment when padding values (see table below), and can be preceded by a *fill* character. A fill cannot be used on its own.
  - The *sign* flag controls the display of signs on numbers (see table below).
  - The `#` flag adds a leading `0b`, `0o`, or `0x` for binary, octal, and hex conversions.

  - The `0` flag zero-pads numbers, equivalent to having a *fill-align* of `0=`.
  - The *width* is a number giving the minimum field width. Padding will be added according to *align* until this width is achieved.
  - For floating-point conversions, *precision* gives the number of places to display after the decimal point. For non-numeric conversion, *precision* gives the maximum field width.
  - The *type* specifies how to present numeric types (see tables below).
- Braces can be doubled (`{{` or `}}`) to insert a literal brace character.

Alignment flag characters

Flag	Meaning
<	Left-aligns the field and pads to the right (default for non-numbers)
>	Right-aligns the field and pads to the left (default for numbers)
=	Inserts padding between the sign and the field (numbers only)
^	Aligns the field to the center and pads both sides

Sign flag characters

Flag	Meaning
+	Displays a sign for all numbers
-	Displays a sign for negative numbers only (default)
	(a space) Displays a sign for negative numbers and a space for positive numbers

Integer type flags

Flag	Meaning
b	Binary format (base 2)
c	Character (interprets integer as a Unicode code point)
d	Decimal format (base 10) (default)
o	Octal format (base 8)
x	Hexadecimal format (base 16) (lowercase)
X	Hexadecimal format (base 16) (uppercase)

Floating-point type flags

Flag	Meaning
e	Exponential format (lowercase)
E	Exponential format (uppercase)
f	Fixed-point format
F	Fixed-point format (same as "f")
g	General format - same as "e" if exponent is greater than -4 or less than precision, "f" otherwise. (default)
G	General format - Same as "E" if exponent is greater than -4 or less than precision, "F" otherwise.
n	Number format - Same as "g", except it uses locale settings for separators.
%	Percentage - Multiplies by 100 and displays as "f", followed by a percent sign.

File objects

(Type `file`). Created with built-in functions **`open()`** [preferred] or its alias **`file()`**. May be created by other modules' functions as well.
Unicode file names are now supported for all functions accepting or returning file names (open, os.listdir, etc...).

Operators on file objects

File operations

Operation	Result
f.**close**()	Close file *f*.
f.**fileno**()	Get fileno (fd) for file *f*.
f.**flush**()	Flush file *f*'s internal buffer.
f.**isatty**()	1 if file *f* is connected to a tty-like dev, else 0.
f.**next**()	Returns the next input line of file *f*, or raises `StopIteration` when EOF is hit. Files are their own *iterators*. `next` is implicitly called by constructs like `for line in f: print line`.
f.**read**([*size*])	Read at most *size* bytes from file *f* and return as a string object. If *size* omitted, read to EOF.
f.**readline**()	Read one entire line from file *f*. The returned line has a trailing \n, except possibly at EOF. Return '' on EOF.
f.**readlines**()	Read until EOF with **readline**() and return a list of lines read.
f.**xreadlines**()	Return a sequence-like object for reading a file line-by-line without reading the

	entire file into memory. From 2.2, use rather: **for** line **in** f (see below).
for line **in** *f***:** do something...	Iterate over the lines of a file (using readline)
f.**seek**(*offset***[**, *whence=0***]**)	Set file *f*'s position, like "stdio's fseek()". *whence* == 0 then use absolute indexing. *whence* == 1 then offset relative to current pos. *whence* == 2 then offset relative to file end.
f.**tell**()	Return file *f*'s current position (byte offset).
f.**truncate**([*size*])	Truncate *f*'s size. If *size* is present, f is truncated to (at most) that size, otherwise f is truncated at current position (which remains unchanged).
f.**write**(*str*)	Write string to file *f*.
f.**writelines**(*list*)	Write list of strings to file *f*. No EOL are added.

File Exceptions

`EOFError`
End-of-file hit when reading (may be raised many times, e.g. if *f* is a tty).

`IOError`
Other I/O-related I/O operation failure

Sets

Since 2.4, Python has 2 new *built-in types* with fast C implementations [PEP 218]: `set` and `frozenset` (immutable set). Sets are unordered collections of unique (non duplicate) elements. Elements must be hashable. `frozensets` are hashable (thus can be elements of other sets) while `sets` are not. All sets are *iterable*.

Since 2.3, the *classes* `Set` and `ImmutableSet` were available in the module `sets`.This module remains in the 2.4 std library in addition to the built-in types.

Main Set operations

Operation	**Result**
set/**frozenset**([*iterable*=None])	[using built-in types] Builds a `set` or `frozenset` from the given *iterable* (default: empty), e.g. `set([1,2,3])`, `set("hello")`.
Set/**ImmutableSet**([*iterable*=None])	[using the `sets` module] Builds a `Set` or `ImmutableSet` from the given *iterable* (default: empty), e.g. `Set([1,2,3])`.
len(*s*)	Cardinality of set *s*.
elt **in** *s* / **not in** *s*	`True` if element *elt* belongs / does not belong to set *s*.
for *elt* **in** *s*: *process elt...*	Iterates on elements of set *s*.
s1.**issubset**(*s2*)	`True` if every element in *s1* is in iterable *s2*.
s1.**issuperset**(*s2*)	`True` if every element in *s2* is in iterable *s1*.
s.**add**(*elt*)	Adds element *elt* to set *s* (if it doesn't already exist).
s.**remove**(*elt*)	Removes element *elt* from set *s*. `KeyError` if element not found.
s.**discard**(*elt*)	Removes element *elt* from set *s* if present.
s.**pop**()	Removes and returns an arbitrary element from set *s*; raises `KeyError` if empty.
s.**clear**()	Removes all elements from this set (not on immutable sets!).
s1.**intersection**(*s2*[, *s3*...]) or *s1***&***s2*	Returns a new Set with elements **common** to all sets (in the method s2, s3,... can be any iterable).
s1.**union**(*s2*[, *s3*...]) or *s1***|***s2*	Returns a new Set with elements from **either** set (in the method s2, s3,... can be any iterable).
s1.**difference**(*s2*[, *s3*...]) or *s1***-***s2*	Returns a new Set with elements in *s1* but not in any of *s2*, *s3* ... (in the method s2, s3,... can be any iterable)
s1.**symmetric_difference**(*s2*) or *s1***^***s2*	Returns a new Set with elements in either *s1* or *s2* but not both.
s.**copy**()	Returns a shallow copy of set *s*.
s.**update**(*iterable1*[, *iterable2*...])	Adds all values from all given iterables to set *s*.

Named Tuples

Python 2.6 module `collections` introduces the `namedtuple` datatype. The factory function `namedtuple(typename, fieldnames)` creates **subclasses** of `tuple` whose fields are accessible **by name** as well as **index**:

```
# Create a named tuple class 'person':
person = collections.namedtuple('person', 'name firstName age') # field names separated by space or comma
assert issubclass(person, tuple)
assert person._fields == ('name', 'firstName', 'age')
```

```
# Create an instance of person:
jdoe = person('Doe', 'John', 30)
assert  str(jdoe) == "person(name='Doe', firstName='John', age=30)"
assert jdoe[0] == jdoe.name == 'Doe' # access by index or name is equivalent
assert jdoe[2] == jdoe.age == 30

# Convert instance to dict:
assert jdoe._asdict() == {'age': 30, 'name': 'Doe', 'firstName': 'John'}

# Although tuples are normally immutable, one can change field values via _replace():
jdoe._replace(age=25, firstName='Jane')
assert str(jdoe) == "person(name='Doe', firstName='Jane', age=25)"
```

Date/Time

Python **has no** intrinsic Date and Time types, but provides 2 built-in modules:

- `time`: time access and conversions
- `datetime`: classes `date`, `time`, `datetime`, `timedelta`, `tzinfo`.

See also the third-party module: `mxDateTime`.

Advanced Types

- See manuals for more details -

- *Module* objects
- *Class* objects
- *Class instance* objects
- *Type* objects (see module: types)
- *File* objects (see above)
- *Slice* objects
- *Ellipsis* object, used by extended slice notation (unique, named `Ellipsis`)
- *Null* object (unique, named `None`)
- *XRange* objects
- **Callable** types:
    - User-defined (written in Python):
        - User-defined *Function* objects
        - User-defined *Method* objects
    - Built-in (written in C):
        - Built-in *Function* objects
        - Built-in *Method* object
- **Internal** Types:
    - *Code* objects (byte-compile executable Python code: *bytecode*)
    - *Frame* objects (execution frames)
    - *Traceback* objects (stack trace of an exception)

Statements

Statement	Result
pass	Null statement
del *name*[, *name*]*	Unbind *name*(s) from object. Object will be indirectly (and automatically) deleted only if no longer referenced.
print[**>>** *fileobject,*] [*s1* [, *s2*]* [,]	Writes to sys.stdout, or to *fileobject* if supplied. Puts spaces between arguments. Puts newline at end unless statement ends with **comma** [if nothing is printed when using a comma, try calling `system.out.flush()`]. Print is not required when running interactively, simply typing an expression will print its value, unless the value is `None`.
exec *x* [in *globals* [, *locals*]]	Executes *x* in namespaces provided. Defaults to current

	namespaces. *x* can be a string, open file-like object or a function object. *locals* can be any mapping type, not only a regular Python dict. See also built-in function `execfile`.
*callable***(***value*,... [*id=value*] , [**args*], [***kw*]**)**	Call function *callable* with parameters. Parameters can be passed by name or be omitted if function defines default values. E.g. if *callable* is defined as "`def callable(p1=1, p2=2)`" "*callable*()" <=> "*callable*(1, 2)" "*callable*(10)" <=> "*callable*(10, 2)" "*callable*(p2=99)" <=> "*callable*(1, 99)" **args* is a tuple of **positional** arguments. ***kw* is a dictionary of **keyword** arguments.

Assignment operators

Assignment operators

Operator	Result	Notes
a **=** *b*	Basic assignment - assign object *b* to label *a*	(1)(2)
a **+=** *b*	Roughly equivalent to *a* = *a* + *b*	(3)
a **-=** *b*	Roughly equivalent to *a* = *a* - *b*	(3)
a ***=** *b*	Roughly equivalent to *a* = *a* * *b*	(3)
a **/=** *b*	Roughly equivalent to *a* = *a* / *b*	(3)
a **//=** *b*	Roughly equivalent to *a* = *a* // *b*	(3)
a **%=** *b*	Roughly equivalent to *a* = *a* % *b*	(3)
a ****=** *b*	Roughly equivalent to *a* = *a* ** *b*	(3)
a **&=** *b*	Roughly equivalent to *a* = *a* & *b*	(3)
a **|=** *b*	Roughly equivalent to *a* = *a* | *b*	(3)
a **^=** *b*	Roughly equivalent to *a* = *a* ^ *b*	(3)
a **>>=** *b*	Roughly equivalent to *a* = *a* >> *b*	(3)
a **<<=** *b*	Roughly equivalent to *a* = *a* << *b*	(3)

Notes:

- (1) Can unpack tuples, lists, and strings:

  ```
  first, second = l[0:2]    # equivalent to: first=l[0]; second=l[1]
  [f, s] = range(2)    # equivalent to: f=0; s=1
  c1,c2,c3 = 'abc'    # equivalent to: c1='a'; c2='b'; c3='c'
  (a, b), c, (d, e, f) = ['ab', 'c', 'def']    # equivalent to: a='a'; b='b'; c='c'; d='d'; e='e';
  f='f'
  ```

  Tip: `x,y = y,x` swaps *x* and *y*.
- (2) Multiple assignment possible:

  ```
  a = b = c = 0
  list1 = list2 = [1, 2, 3]    # list1 and list2 points to the same list (l1 is l2)
  ```

- (3) Not exactly equivalent - *a* is evaluated only once. Also, where possible, operation performed in-place - *a* is modified rather than replaced.

Conditional Expressions

Conditional *Expressions* (not *statements*) have been added since 2.5 [PEP 308]:

```
result = (whenTrue if condition else whenFalse)
```

is equivalent to:

```
if condition:
    result = whenTrue
else:
    result = whenFalse
```

() are not mandatory but recommended.

Control Flow statements

Control flow statements

Statement	Result
if *condition***:** *suite* [**elif** *condition***:** *suite*]* [**else:** *suite*]	Usual if/else if/else statement. See also Conditional Expressions.
while *condition***:** *suite* [**else:** *suite*]	Usual while statement. The `else` *suite* is executed after loop exits, unless the loop is exited with `break`.
for *element* **in** *sequence***:** *suite* [**else:** *suite*]	Iterates over *sequence*, assigning each element to *element*. Use built-in `range` function to iterate a number of times. The `else` *suite* is executed at end unless loop exited with `break`.
break	Immediately exits `for` or `while` loop.
continue	Immediately does next iteration of `for` or `while` loop.
return *[result]*	Exits from function (or method) and returns *result* (use a **tuple** to return more than one value). If no result given, then returns `None`.
yield *expression*	(Only used within the body of a generator function, outside a try of a `try..finally`). "Returns" the evaluated *expression*.

Exception statements

Exception statements

Statement	Result
assert *expr*[, *message*]	*expr* is evaluated. if false, raises exception `AssertionError` with message. Before 2.3, inhibited if `__debug__` is 0.
try: *block1* [**except** [*exception* [, *value*]]**:** *handler*]+ [**except** [*exception* [as *value*]]**:** *handler*]+ [**else:** *else-block*]	Statements in *block1* are executed. If an exception occurs, look in `except` clause(s) for matching *exception(s)*. If matches or bare `except`, execute *handler* of that clause. If no exception happens, *else-block* in `else` clause is executed after *block1*. If *exception* has a value, it is put in variable *value*. *exception* can also be a **tuple** of exceptions, e.g. `except(KeyError, NameError), e: print e`. 2.6 also supports the keyword `as` instead of a comma between the *exception* and the *value*, which will become a mandatory change in Python 3.0 [PEP3110].
try: *block1* **finally:** *final-block*	Statements in *block1* are executed. If no exception, execute *final-block* (even if *block1* is exited with a `return`, `break` or `continue` statement). If exception did occur, execute *final-block* and then immediately re-raise exception. Typically used to ensure that a resource (file, lock...) allocated before the `try` is freed (in the `final-block`) whatever the outcome of `block1` execution. See also the with statement below.
try: *block1* [**except** [*exception* [, *value*]]**:** *handler1*]+ [**except** [*exception* [as *value*]]**:** *handler*]+ [**else:** *else-block*] **finally:** *final-block*	Unified try/except/finally. Equivalent to a `try...except` nested inside a `try..finally` [PEP341]. See also the with statement below.
with *allocate-expression* [**as** *variable*] *with-block*	Alternative to the `try...finally` structure [PEP343]. *allocate-expression* should evaluate to an object that supports the *context management protocol*, representing a resource. This object may return a value that can optionally be bound to *variable* (variable is **not** assigned the result of expression). The object can then run **set-up** code before `with-block` is executed and some **clean-up** code is executed after the block is

	done, even if the block raised an exception. Standard Python objects such as files and locks support the context management protocol: **with** open('/etc/passwd', 'r') **as** f: # file automatically closed on block exit for line in f: print line **with** threading.Lock(): # lock automatically released on block exit do something... - You can write your own context managers. - Helper functions are available in module contextlib. In 2.5 the statement must be enabled by: `from __future__ import with_statement`. The statement is always enabled starting in Python 2.6.
raise *exceptionInstance*	Raises an instance of a class derived from `BaseException` (**preferred** form of raise).
raise *exceptionClass* [, *value* [, *traceback*]]	Raises *exception* of given class *exceptionClass* with optional value *value*. Arg *traceback* specifies a traceback object to use when printing the exception's backtrace.
raise	A raise statement without arguments re-raises the last exception raised in the current function.

- An exception is an *instance* of an *exception class* (before 2.0, it may also be a mere *string*).
- Exception classes must be derived from the predefined class: Exception, e.g.:

```
class TextException(Exception): pass
try:
    if bad:
        raise TextException()
except Exception:
    print 'Oops'  # This will be printed because TextException is a subclass of Exception
```

- When an error message is printed for an unhandled exception, the class name is printed, then a colon and a space, and finally the instance converted to a string using the built-in function `str()`.
- All built-in exception classes derives from `StandardError`, itself derived from `Exception`.
- [PEP 352]: Exceptions can now be **new-style classes**, and all built-in ones are. Built-in exception hierarchy slightly reorganized with the introduction of base class `BaseException`. Raising strings as exceptions is now deprecated (warning).

Name Space Statements

Imported module files must be located in a directory listed in the Python path (`sys.path`). Since 2.3, they may reside in a **zip** file [e.g. sys.path.insert(0, "aZipFile.zip")].

Absolute/relative imports (since 2.5 [PEP328]):

- Feature must be enabled by: `from __future__ import absolute_import`: will probably be adopted in 2.7.
- Imports are normally *relative*: modules are searched first in the current directory/package, and then in the builtin modules, resulting in possible ambiguities (e.g. masking a builtin symbol).
- When the new feature is enabled:
  - `import X` will look up for module X in `sys.path` first (*absolute* import).
  - `import .X` (with a dot) will still search for X in the current package first, then in builtins (*relative* import).
  - `import ..X` will search for X in the package containing the current one, etc...

Packages (>1.5): a **package** is a name space which maps to a directory including module(s) and the special initialization module `__init__.py` (possibly empty).
Packages/directories can be nested. You address a module's symbol via `[package.[package...].module.symbol`.
[1.51: On Mac & Windows, the case of module file names must now match the case as used in the *import* statement]

Name space statements

Statement	Result

import *module1* [**as** *name1*] [, *module2*]*	Imports modules. Members of module must be referred to by qualifying with [package.]module name, e.g.:
	`import sys; print sys.argv`
	`import package1.subpackage.module`
	`package1.subpackage.module.foo()`

	module1 renamed as *name1*, if supplied.

from *module* **import** *name1* [**as** *othername1*][, *name2*]*	Imports names from module *module* in current namespace.
	`from sys import argv; print argv`
	`from package1 import module; module.foo()`
	`from package1.module import foo; foo()`
	name1 renamed as *othername1*, if supplied.
	[2.4] You can now put parentheses around the list of names in a `from module import names` statement (PEP 328).

from *module* **import ***	Imports **all** names in *module*, except those starting with "_". **Use sparsely, beware of name clashes!**
	`from sys import *; print argv`
	`from package.module import *; print x`
	Only legal at the top level of a module.
	If *module* defines an `__all__` attribute, only names listed in `__all__` will be imported.
	NB: "`from package import *`" only imports the symbols defined in the package's `__init__.py` file, not those in the package's modules !

global *name1 [, name2]*	Names are from global scope (usually meaning from module) rather than local (usually meaning only in function). E.g. in function without `global` statements, assuming "x" is name that hasn't been used in function or module so far: - Try to read from "x" -> `NameError` - Try to write to "x" -> creates "x" local to function If "x" not defined in fct, but is in module, then: - Try to read from "x", gets value from module - Try to write to "x", creates "x" local to fct But note "x[0]=3" starts with search for "x", will use to global "x" if no local "x".

Function Definition

```
def funcName ([paramList]):
      suite
```

Creates a function object and binds it to name *funcName*.

```
paramList ::= [param [, param]*]
param ::= value | id=value | *id | **id
```

- Args are passed by **value**, so only args representing a *mutable* object can be modified (are *inout* parameters).
- Use `return` to return (`None`) from the function, or `return value` to return *value*. Use a **tuple** to return more than one value, e.g. `return 1,2,3`
- *Keyword* arguments `arg=value` specify a *default value* (evaluated at function def. time). They can only appear **last** in the param list, e.g. `foo(x, y=1, s='')`.
- Pseudo-arg ***args** captures a tuple of all remaining non-keyword args passed to the function, e.g. if `def foo(x, *args): ...` is called `foo(1, 2, 3)`, then *args* will contain `(2,3)`.
- Pseudo-arg ****kwargs** captures a dictionary of all extra keyword arguments, e.g. if `def foo(x, **kwargs): ...` is called `foo(1, y=2, z=3)`, then *kwargs* will contain `{'y':2, 'z':3}`. if `def foo(x, *args, **kwargs): ...` is called `foo(1, 2, 3, y=4, z=5)`, then *args* will contain `(2, 3)`, and *kwargs* will contain `{'y':4, 'z':5}`
- *args* and *kwargs* are conventional names, but other names may be used as well.
- **args* and ***kwargs* can be "forwarded" (individually or together) to another function, e.g.
  ```
  def f1(x, *args, **kwargs):
    f2(*args, **kwargs)
  ```
  Since 2.6, **kwargs can be any mapping, not only a `dict`.
- See also Anonymous functions (*lambdas*).

Class Definition

```
class className [(super_class1[, super_class2]*)]:
  suite
```

Creates a class object and assigns it name *className*.
suite may contain local "defs" of class methods and assignments to class attributes.

Examples:

class MyClass (class1, class2): ...
Creates a class object inheriting from both class1 and class2. Assigns new class object to name `MyClass`.

class MyClass: ...
Creates a *base* class object (inheriting from nothing). Assigns new class object to name `MyClass`. Since 2.5 the equivalent syntax **class** `MyClass(): ...` is allowed.

class MyClass (**object**): ...
Creates a *new-style* class (inheriting from `object` makes a class a *new-style* class -available since Python 2.2-). Assigns new class object to name `MyClass`.

- First arg to class instance methods (operations) is always the target instance object, called '**self**' by convention.
- Special static method **__new__**(*cls*[,...]) called when instance is created. 1st arg is a class, others are args to __init__(), more details here
- Special method **__init__**() is called when instance is created.
- Special method **__del__**() called when no more reference to object.
- Create instance by "*calling*" class object, possibly with arg (thus instance=apply(aClassObject, args...) creates an instance!)
- Before 2.2 it was not possible to subclass built-in classes like list, dict (you had to "wrap" them, using UserDict & UserList modules); since 2.2 you can subclass them directly (see Types/Classes unification).

Example:

```
class c (c_parent):
    def __init__(self, name):
        self.name = name
    def print_name(self):
        print "I'm", self.name
    def call_parent(self):
        c_parent.print_name(self)

instance = c('tom')
print instance.name
'tom'
instance.print_name()
"I'm tom"
```

Call parent's super class by accessing parent's method directly and passing `self` explicitly (see `call_parent` in example above).
Many other special methods available for implementing arithmetic operators, sequence, mapping indexing, etc...

Types / classes unification

Base types `int`, `float`, `str`, `list`, `tuple`, `dict` and `file` now (2.2) behave like **classes** derived from base class `object`, and may be **subclassed**:

```
x = int(2) # built-in cast function now a constructor for base type
y = 3  # <=> int(3) (litterals are instances of new base types)
print type(x), type(y) # int, int
assert isinstance(x, int) # replaces isinstance(x, types.IntType)
assert issubclass(int, object) # base types derive from base class 'object'.
s = "hello" # <=> str("hello")
assert isinstance(s, str)
f = 2.3  # <=> float(2.3)
class MyInt(int): pass    # may subclass base types
x,y = MyInt(1), MyInt("2")
print x, y, x+y # => 1,2,3
class MyList(list): pass
l = MyList("hello")
print l # ['h', 'e', 'l', 'l', 'o']
```

New-style classes extends `object`. *Old-style* classes don't.

Documentation Strings

Modules, classes and functions may be documented by placing a string literal by itself as the first statement in the suite. The documentation can be retrieved by getting the '**__doc__**' attribute from the module, class or function.

Example:

```
class C:
    "A description of C"
        def __init__(self):
```

```
            "A description of the constructor"
            # etc.

c.__doc__ == "A description of C".
c.__init__.__doc__ == "A description of the constructor"
```

Iterators

- An *iterator* enumerates elements of a *collection*. It is an object with a single method `next()` returning the next element or raising `StopIteration`.
- You get an iterator on *obj* via the new built-in function `iter(obj)`, which calls `obj.__class__.__iter__()`.
- A collection may be its **own** iterator by implementing both `__iter__()` and `next()`.
- Built-in collections (lists, tuples, strings, dict) implement `__iter__()`; dictionaries (maps) enumerate their keys; files enumerates their lines.
- You can build a `list` or a `tuple` from an iterator, e.g. `list(anIterator)`
- Python implicitly uses iterators wherever it has to **loop** :
  - **for** elt **in** *collection*:
  - **if** elt **in** *collection*:
  - when assigning tuples: `x,y,z=` *collection*

Generators

- A *generator* is a function that retains its state between 2 calls and produces a **new** value at **each** invocation. The values are returned (one at a time) using the keyword `yield`, while `return` or raise `StopIteration()` are used to notify the end of values.
- A typical use is the production of IDs, names, or serial numbers. Fancier applications like nanothreads are also possible.
- In 2.2, the feature needs to be **enabled** by the statement: `from __future__ import generators` (not required since 2.3+)
- To **use** a generator: call the *generator function* to get a generator object, then call `generator.next()` to get the next value until `StopIteration` is raised.
- 2.4 introduces ***generator expressions*** [PEP 289] similar to list comprehensions, except that they create a generator that will return elements one by one, which is suitable for long sequences :
  ```
  linkGenerator = (link for link in get_all_links() if not link.followed)
  for link in linkGenerator:
    ...process link...
  ```
  Generator expressions must appear between **parentheses**.
- [PEP342] Generators before 2.5 could only produce **output**. Now values can be **passed** to generators via their method `send(value)`. `yield` is now an *expression* returning a value, so `val = (yield i)` will *yield* i to the caller, and will reciprocally evaluate to the value "sent" back by the caller, or `None`.
  Two other new generator methods allow for additional control:
  - `throw(type, value=None, traceback=None)` is used to raise an exception inside the generator (appears as raised by the `yield` expression).
  - `close()` raises a new `GeneratorExit` exception inside the generator to terminate the iteration.
- Since 2.6 Generator objects have a `gi_code` attribute that refers to the original code object backing the generator.

Example:

```
def genID(initialValue=0):
  v = initialValue
  while v < initialValue + 1000:
    yield "ID_%05d" % v
    v += 1
  return    # or: raise StopIteration()
generator = genID() # Create a generator
for i in range(10): # Generates 10 values
  print generator.next()
```

Descriptors / Attribute access

- *Descriptors* are objects implementing at least the first of these 3 methods representing the *descriptor protocol*:
  - `__get__(self, obj, type=None) --> value`
  - `__set__(self, obj, value)`
  - `__delete__(self, obj)`

  Python now transparently uses *descriptors* to describe and access the attributes and methods of new-style classes (i.e. derived from `object`).)

- Built-in descriptors now allow to define:
  - **Static methods** : Use `staticmethod(f)` to make method `f(x)` static (unbound).
  - **Class methods**: like a static but takes the Class as 1st argument => Use `f = classmethod(f)` to make method `f(theClass, x)` a class method.
  - **Properties** : A *property* is an instance of the new built-in type `property`, which implements the *descriptor* protocol for attributes => Use `propertyName = property(fget=None, fset=None, fdel=None, doc=None)` to define a property inside or outside a class. Then access it as `propertyName` or `obj.propertyName`.
    Since 2.6, the new decorators `@prop.getter`, `@prop.setter`, and `@prop.deleter` add functions to an existing property:

```
class C(object):
    @property # (since Python 2.4)
    def x(self):
        return self._x

    @x.setter
    def x(self, value):
        self._x = value

    @x.deleter
    def x(self):
        del self._x
```

  - **Slots**. New style classes can define a class attribute `__slots__` to constrain the list of **assignable** attribute names, to avoid typos (which is normally not detected by Python and leads to the creation of new attributes), e.g. `__slots__ = ('x', 'y')`
    Note: According to recent discussions, the real purpose of slots seems still unclear (optimization?), and their use should probably be discouraged.

Decorators for functions, methods & classes

- [PEP 318] A *decorator* D is noted `@D` on the line preceding the function/method it decorates :
```
@D
def f(): ...
```
  and is equivalent to:
```
def f(): ...
f = D(f)
```
- Several decorators can be applied in cascade :
```
@A
@B
@C
def f(): ...
```
  is equivalent to:
```
f = A(B(C(f)))
```
- A decorator is just a function taking the fct to be decorated and returns the same function or some new callable thing.
- Decorator functions can take arguments:
```
@A
@B
@C(args)
```
  becomes :
```
def f(): ...
_deco = C(args)
f = A(B(_deco(f)))
```
- The decorators `@staticmethod` and `@classmethod` replace more elegantly the equivalent declarations `f = staticmethod(f)` and `f = classmethod(f)`.
- [PEP 3129] Decorators may also be applied to classes:
```
@D
class C(): ...
```
  is equivalent to:
```
class C(): ...
C = D(C)
```

Misc

lambda [*param_list*]: *returnedExpr*

Creates an **anonymous** function.

returnedExpr must be an expression, not a statement (e.g., not "if xx:...", "print xxx", etc.) and thus can't contain newlines. Used mostly for filter(), map(), reduce() functions, and GUI callbacks.

List comprehensions

```
result = [expression for item1 in sequence1  [if condition1]
             [for item2 in sequence2 ... for itemN in sequenceN]
         ]
```

is equivalent to:

```
result = []
for item1 in sequence1:
    for item2 in sequence2:
        ...
        for itemN in sequenceN:
            if (condition1) and further conditions:
               result.append(expression)
```

See also Generator expressions.

Nested scopes
Since 2.2 *nested scopes* no longer need to be specially enabled by a `from __future__ import nested_scopes` directive, and are always used.

Built-In Functions

Built-in functions are defined in a module `__builtin__` automatically imported.

Built-In Functions

Function	Result
__import__(*name*[, *globals*[,*locals*[,*from list*]]])	Imports module within the given context (see library reference for more details)
abs(*x*)	Returns the absolute value of the number *x*.
all(*iterable*)	Returns `True` if `bool(x)` is `True` for **all** values x in the iterable.
any(*iterable*)	Returns `True` if `bool(x)` is `True` for **any** value x in the iterable.
apply(*f*, *args*[, *keywords*])	Calls func/method *f* with arguments *args* and optional keywords. deprecated since 2.3, replace `apply(func, args, keywords)` with `func(*args, **keywords)` [details]
basestring()	Abstract superclass of `str` and `unicode`; can't be called or instantiated directly, but useful in: `isinstance(obj, basestring)`.
bin(*x*)	Converts a number to a binary string.
bool([*x*])	Converts a value to a Boolean, using the standard truth testing procedure. If *x* is false or omitted, returns `False`; otherwise returns `True`. bool is also a class/type, subclass of `int`. Class `bool` cannot be subclassed further. Its only instances are `False` and `True`. See also boolean operators
buffer(*object*[, *offset*[, *size*]])	Returns a `Buffer` from a slice of *object*, which must support the buffer call interface (string, array, buffer). *Non essential function*, see [details]
bytearray(*iterable*) **bytearray**(*length*)	Constructs a mutable sequence of `bytes`. This type supports many of the same operations available in `str`s and `list`s. The latter form sets the size and initializes to all zero bytes.
bytes(*object*)	Constructs an 8-bit string representation of an object. Equivalent to `str` for now, but this can be used to explicitly indicate strings which should not be unicode when converting to Python 3.0 [PEP3112]
callable(*x*)	Returns True if *x* callable, else False.
chr(*i*)	Returns one-character string whose ASCII code is integer *i*.

classmethod (*function*)	Returns a class method for *function*. A class method receives the class as implicit first argument, just like an instance method receives the instance. To declare a class method, use this idiom:
	
	`class C:`
	`   def f(cls, arg1, arg2, ...): ...`
	`   f = classmethod(f)`
	
	Then call it on the class `C.f()` or on an instance `C().f()`. The instance is ignored

	except for its class. If a class method is called for a derived class, the derived class object is passed as the implied first argument.
	Since 2.4 you can alternatively use the decorator notation:
	`class C:`
	    `@classmethod`
	    `def f(cls, arg1, arg2, ...): ...`

cmp(*x*,*y*)	Returns negative, 0, positive if *x* <, ==, > to *y* respectively.
coerce(*x*,*y*)	Returns a tuple of the two *numeric* arguments converted to a common type. *Non essential function*, see [details]
compile(*string*, *filename*, *kind[, flags[, dont_inherit]]*)	Compiles *string* into a code object. *filename* is used in error message, can be any string. It is usually the file from which the code was read, or e.g. `'<string>'` if not read from file. *kind* can be **'eval'** if *string* is a single stmt, or **'single'** which prints the output of expression statements that evaluate to something else than None, or be **'exec'**. New args *flags* and *dont_inherit* concern *future* statements. Since 2.6 the fct accepts **keyword** arguments as well as positional parameters.
complex(*real*[, *image*])	Creates a `complex` object (can also be done using **J** or **j** suffix, e.g. `1+3J`). Since 2.6, also accepts strings, with or without parenthesis, e.g. `complex('1+3J')` or `complex('(1+3J)')`.
delattr(*obj*, *name*)	Deletes the attribute named *name* of object *obj* <=> `del obj.name`
dict([*mapping-or-sequence*])	Returns a new dictionary initialized from the optional argument (or an empty dictionary if no argument). Argument may be a sequence (or anything iterable) of pairs (key,value).
dir([*object*])	Without args, returns the list of names in the current local symbol table. With a module, class or class instance object as *arg*, returns the list of names in its attr. dictionary. Since 2.6 *object* can override the std implementation via special method `__dir__()`.
divmod(*a*,*b*)	Returns tuple (*a*//*b*, *a*%*b*)
enumerate(*iterable*)	Iterator returning pairs (index, value) of *iterable*, e.g. `List(enumerate('Py')) -> [(0, 'P'), (1, 'y')]`.
eval(*s*[, *globals*[, *locals*]])	Evaluates string *s*, representing a single python *expression*, in (optional) *globals*, *locals* contexts. *s* must have no NUL's or newlines. *s* can also be a code object. *locals* can be any mapping type, not only a regular Python dict. Example: `x = 1; assert eval('x + 1') == 2` (To execute *statements* rather than a single expression, use Python statement `exec` or built-in function `execfile`)
execfile(*file*[, *globals* [,*locals*]])	Executes a file without creating a new module, unlike `import`. *locals* can be any mapping type, not only a regular Python dict.
file(*filename*[,*mode* [,*bufsize*]])	Opens a file and returns a new `file` object. Alias for `open`.
filter (*function*,*sequence*)	Constructs a list from those elements of *sequence* for which *function* returns true. *function* takes one parameter.
float(*x*)	Converts a number or a string to floating point. Since 2.6, *x* can be one of the strings `'nan'`, `'+inf'`, or `'-inf'` to represent respectively IEEE 754 Not A Number, positive and negative infinity. Use module `math` fcts `isnan()` and `isinf()` to check for NAN or infinity.
format(*value*[, *format_spec*])	Formats an object with the given specification (default '') by calling its `__format__` method.
frozenset([*iterable*])	Returns a `frozenset` (immutable set) object whose (immutable) elements are taken from *iterable*, or empty by default. See also Sets.
getattr(*object*,*name* [,*default*]))	Gets attribute called *name* from *object*, e.g. getattr(x, 'f') <=> x.f). If not found, raises `AttributeError` or returns *default* if specified.
globals()	Returns a dictionary containing the current global variables.
hasattr(*object*, *name*)	Returns true if *object* has an attribute called *name*.
hash(*object*)	Returns the hash value of the object (if it has one).
help([*object*])	Invokes the built-in help system. No argument -> interactive help; if *object* is a string (**name** of a module, function, class, method, keyword, or documentation topic), a help page is printed on the console; otherwise a help page on *object* is generated.
hex(*x*)	Converts a number *x* to a hexadecimal string.
id(*object*)	Returns a unique integer identifier for *object*. Since 2.5 always returns non-negative numbers.
input([*prompt*])	Prints *prompt* if given. Reads input and **evaluates** it. Uses line editing / history if module `readline` available.

int(*x*[, *base*])	Converts a number or a string to a plain integer. Optional *base* parameter specifies base from which to convert string values.
intern(*aString*)	Enters *aString* in the table of interned strings and returns the string. Since 2.3, interned strings are no longer 'immortal' (never garbage collected), see [details]
isinstance(*obj*, *classInfo*)	Returns true if *obj* is an instance of **class** *classInfo* or an object of **type** *classInfo* (*classInfo* may also be a **tuple** of classes or types). If `issubclass(A,B)` then `isinstance(x,A)` => `isinstance(x,B)`
issubclass(*class1*, *class2*)	Returns true if *class1* is derived from *class2* (or if *class1* `is` *class2*).
iter(*obj*[,*sentinel*])	Returns an **iterator** on *obj*. If *sentinel* is absent, *obj* must be a collection implementing either `__iter__()` or `__getitem__()`. If *sentinel* is given, *obj* will be **called** with no arg; if the value returned is equal to *sentinel*, `StopIteration` will be raised, otherwise the value will be returned. See Iterators.
len(*obj*)	Returns the length (the number of items) of an object (sequence, dictionary, or instance of class implementing __len__).
list([*seq*])	Creates an empty list or a list with same elements as *seq*. *seq* may be a sequence, a container that supports iteration, or an iterator object. If *seq* is already a list, returns a **copy** of it.
locals()	Returns a dictionary containing current local variables.
long(*x*[, *base*])	Converts a number or a string to a long integer. Optional *base* parameter specifies the base from which to convert string values.
map(*function*, *sequence*[, *sequence*, ...])	Returns a list of the results of applying *function* to each item from *sequence*(s). If more than one sequence is given, the function is called with an argument list consisting of the corresponding item of each sequence, substituting `None` for missing values when not all sequences have the same length. If *function* is `None`, returns a list of the items of the sequence (or a list of tuples if more than one sequence). => You might also **consider using list comprehensions instead of map()**.
max(*iterable*[, *key*=func]) **max**(*v1*, *v2*, ...[, *key*=func])	With a single argument *iterable*, returns the **largest** item of a non-empty iterable (such as a string, tuple or list). With more than one argument, returns the largest of the arguments. The optional *key* arg is a function that takes a single argument and is called for every value in the list.
min(*iterable*[, *key*=func]) **min**(*v1*, *v2*, ...[, *key*=func])	With a single argument *iterable*, returns the **smallest** item of a non-empty iterable (such as a string, tuple or list). With more than one argument, returns the smallest of the arguments. The optional *key* arg is a function that takes a single argument and is called for every value in the list.
next(*iterator*[, *default*])	Returns the next item from *iterator*. If iterator exhausted, returns *default* if specified, or raises `StopIteration` otherwise.
object()	Returns a new featureless object. `object` is the base class for all *new style classes*, its methods are common to all instances of new style classes.
oct(*x*)	Converts a number to an octal string.
open(*filename* [, *mode*='r', [*bufsize*]])	Returns a new file object. See also alias file(). Use `codecs.open()` instead to open an **encoded** file and provide transparent encoding / decoding. • *filename* is the file name to be opened • *mode* indicates how the file is to be opened: ○ 'r' for reading ○ 'w' for writing (truncating an existing file) ○ 'a' opens it for appending ○ '+' (appended to any of the previous modes) open the file for updating (note that 'w+'truncates the file) ○ 'b' (appended to any of the previous modes) open the file in binary mode ○ 'U' (or 'rU') open the file for reading in *Universal Newline mode*: all variants of EOL (CR, LF, CR+LF) will be translated to a single LF ('\n'). • *bufsize* is 0 for unbuffered, 1 for line buffered, negative or omitted for system default, >1 for a buffer of (about) the given size.
ord(*c*)	Returns integer ASCII value of *c* (a string of len 1). Works with Unicode char.
pow(*x*, *y* [, *z*])	Returns *x* to power *y* [modulo *z*]. See also ** operator.

property([*fget*[, *fset*[, *fdel*[, *doc*]]]])	Returns a property attribute for *new-style* classes (classes deriving from `object`). *fget*, *fset*, and *fdel* are functions to get the property value, set the property value, and delete the property, respectively. Typical use:
	`class C(object):`
	`  def __init__(self): self.__x = None`
	`  def getx(self): return self.__x`
	`  def setx(self, value): self.__x = value`
	`  def delx(self): del self.__x`
	`  x = property(getx, setx, delx, "I'm the 'x' property.")`

print(**args*[, *sep*=' '][, *end*='\n'][, *file*=sys.stdout])	When `__future__.print_function` is active, the `print` statement is **replaced** by this function [PEP3105]. Each item in *args* is printed to *file* with *sep* as the delimiter, and finally followed by *end*.
	
	Each of these statements:
	`print 'foo', 42`
	`print 'foo', 42,`
	`print >> sys.stderr 'warning'`
	can now be written in this functional form:
	`print('foo', 42)`
	`print('foo', 42, end='')`
	`print('warning', file=sys.stderr)`

range([*start*,] *end* [, *step*])	Returns list of ints from >= start and < end. With 1 arg, list from 0..*arg*-1 With 2 args, list from *start*..*end*-1 With 3 args, list from *start* up to *end* by *step*
raw_input([*prompt*])	Prints *prompt* if given, then reads string from std input (no trailing \n). See also input().
reduce(*f*, *list* [, *init*])	Applies the binary function *f* to the items of *list* so as to reduce the list to a single value. If *init* is given, it is "prepended" to *list*.
reload(*module*)	Re-parses and re-initializes an already imported module. Useful in interactive mode, if you want to reload a module after fixing it. If module was syntactically correct but had an error in initialization, must import it one more time before calling reload().
repr(*object*)	Returns a string containing a printable and if possible **evaluable** representation of an object. <=> `` `object` `` (using backquotes). Class redefinable (`__repr__`). See also str()
round(*x*, *n*=0)	Returns the floating point value *x* rounded to *n* digits after the decimal point.
set([*iterable*])	Returns a `set` object whose elements are taken from *iterable*, or empty by default. See also Sets.
setattr(*object*, *name*, *value*)	This is the counterpart of getattr().setattr(o, 'foobar', 3) <=> o.foobar = 3. **Creates** attribute if it doesn't exist!
slice([*start*,] *stop*[, *step*])	Returns a *slice object* representing a range, with R/O attributes: start, stop, step.
sorted(*iterable*[, *cmp*[, *key*[, *reverse*]]])	Returns a **new** sorted list from the items in *iterable*. This contrasts with `list.sort()` that sorts lists **in place** and doesn't apply to immutable sequences like strings or tuples. See *sequences*.sort method.

staticmethod (*function*)	Returns a static method for *function*. A static method does not receive an implicit first argument. To declare a static method, use this idiom:
	
	`class C:`
	`    def f(arg1, arg2, ...): ...`
	`    f = staticmethod(f)`
	
	Then call it on the class `C.f()` or on an instance `C().f()`. The instance is ignored except for its class.
	Since 2.4 you can alternatively use the decorator notation:
	`class C:`
	`    @staticmethod`
	`    def f(arg1, arg2, ...): ...`

str(*object*)	Returns a string containing a nicely printable representation of an object. Class overridable (__str__). See also repr().
sum(*iterable*[, *start=0*])	Returns the sum of a sequence of numbers (**not** strings), plus the value of parameter. Returns *start* when the sequence is empty.

super(*type*[, *object-or-type*])	Returns the superclass of *type*. If the second argument is omitted the super object returned is unbound. If the second argument is an object, `isinstance(obj, type)` must be true. If the second argument is a type, `issubclass(type2, type)` must be true. Typical use:
	`class C(B):`
	`  def meth(self, arg):`
	`    super(C, self).meth(arg)`

tuple([*seq*])	Creates an empty tuple or a tuple with same elements as *seq*. *seq* may be a sequence, a container that supports iteration, or an iterator object. If *seq* is already a tuple, returns **itself** (not a copy).
type(*obj*)	Returns a *type object* [see module *types*] representing the type of *obj*. Example: import types if type(x) == types.StringType: print 'It is a string'. NB: it is better to use instead: if isinstance(x, types.StringType)...

unichr(*code*)	Returns a unicode string 1 char long with given *code*.
unicode(*string*[, *encoding*[,*error*]]])	Creates a Unicode string from a 8-bit string, using the given encoding name and error treatment ('strict', 'ignore',or 'replace'}. For objects which provide a `__unicode__()` method, it will call this method without arguments to create a Unicode string.
vars([*object*])	Without arguments, returns a dictionary corresponding to the current local symbol table. With a module, class or class instance object as argument, returns a dictionary corresponding to the object's symbol table. Useful with the "%" string formatting operator.
xrange(*start* [, *end* [, *step*]])	Like range(), but doesn't actually store entire list all at once. Good to use in "for" loops when there is a big range and little memory.
zip(*seq1*[, *seq2*,...])	[No, that's not a compression tool! For that, see module zipfile] Returns a list of tuples where each tuple contains the *n*th element of each of the argument sequences. Since 2.4 returns an empty list if called with no arguments (was raising `TypeError` before).

Built-In Exception classes

BaseException
Mother of all exceptions (was `Exception` before 2.5). New-style class. `*exception*.args` is a tuple of the arguments passed to the constructor.Since 2.6 the `*exception*.message` attribute is deprecated.
`KeyboardInterrupt` & `SystemExit` were moved out of `Exception` because they don't really represent errors, so now a `try:...except Exception:` will only catch **errors**, while a `try:...except BaseException:` (or simply `try:..except:`) will still catch **everything**.

- **GeneratorExit**
  Raised by the `close()` method of *generators* to terminate the iteration. Before 2.6 was derived from `Exception`.
- **KeyboardInterrupt**
  On user entry of the interrupt key (often `CTRL-C'). Before 2.5 was derived from `Exception`.
- **SystemExit**
  On `sys.exit()`. Before 2.5 was derived from `Exception`.
- **Exception**
  Base of all *errors*. Before 2.5 was the base of all exceptions.
  - ~~**GeneratorExit**~~
    Moved under `BaseException`.
  - **StandardError**
    Base class for all built-in exceptions; derived from `Exception` root class.
    - **ArithmeticError**
      Base class for arithmetic errors.
      - **FloatingPointError**
        When a floating point operation fails.
      - **OverflowError**
        On excessively large arithmetic operation.
      - **ZeroDivisionError**
        On division or modulo operation with 0 as 2nd argument.
    - **AssertionError**
      When an *assert* statement fails.
    - **AttributeError**
      On attribute reference or assignment failure
    - **EnvironmentError [new in 1.5.2]**
      On error outside Python; error arg. tuple is (errno, errMsg...)
      - **IOError [changed in 1.5.2]**
        I/O-related operation failure.
      - **OSError [new in 1.5.2]**
        Used by the *os* module's *os.error* exception.
        - **WindowsError**
          When a Windows-specific error occurs or when the error number does not correspond to an `errno` value.
    - **EOFError**
      Immediate end-of-file hit by input() or raw_input()
    - **ImportError**
      On failure of `import` to find module or name.
    - ~~**KeyboardInterrupt**~~

    Moved under `BaseException`.
  - **LookupError**
    base class for IndexError, KeyError
    - **IndexError**
      On out-of-range sequence subscript
    - **KeyError**
      On reference to a non-existent mapping (dict) key
  - **MemoryError**
    On recoverable memory exhaustion
  - **NameError**
    On failure to find a local or global (unqualified) name.
    - **UnboundLocalError**
      On reference to an unassigned local variable.
  - **ReferenceError**
    On attempt to access to a garbage-collected object via a weak reference proxy.
  - **RuntimeError**
    Obsolete catch-all; define a suitable error instead.
    - **NotImplementedError [new in 1.5.2]**
      On method not implemented.
  - **SyntaxError**
    On parser encountering a syntax error
    - **IndentationError**
      On parser encountering an indentation syntax error
      - **TabError**
        On improper mixture of spaces and tabs
  - **SystemError**
    On non-fatal interpreter error - bug - report it !
  - **TypeError**
    On passing inappropriate type to built-in operator or function.
  - **ValueError**
    On argument error not covered by `TypeError` or more precise.
    - **UnicodeError**
      On Unicode-related encoding or decoding error.
      - **UnicodeDecodeError**
        On Unicode decoding error.
      - **UnicodeEncodeError**
        On Unicode encoding error.
      - **UnicodeTranslateError**
        On Unicode translation error.
- **StopIteration**
  Raised by an iterator's `next()` method to signal that there are no further values.
- ~~**SystemExit**~~
  Moved under `BaseException`.
- **Warning**
  Base class for warnings (see module `warning`)
  - **DeprecationWarning**
    Warning about deprecated code.
  - **FutureWarning**
    Warning about a construct that will change semantically in the future.
  - **ImportWarning**
    Warning about probable mistake in module import (e.g. missing `__init__.py`).
  - **OverflowWarning**
    Warning about numeric overflow. Won't exist in Python 2.5.
  - **PendingDeprecationWarning**
    Warning about future deprecated code.
  - **RuntimeWarning**
    Warning about dubious runtime behavior.
  - **SyntaxWarning**
    Warning about dubious syntax.
  - **UnicodeWarning**
    When attempting to compare a Unicode string and an 8-bit string that can't be converted to Unicode using default ASCII encoding (raised a `UnicodeDecodeError` before 2.5).
  - **UserWarning**
    Warning generated by user code.

Standard methods & operators redefinition in classes

Standard methods & operators map to special methods '__*method*__' and thus can be **redefined** (mostly in user-defined classes), e.g.:

```
class C:
    def __init__(self, v): self.value = v
    def __add__(self, r): return self.value + r
a = C(3) # sort of like calling C.__init__(a, 3)
a + 4    # is equivalent to a.__add__(4)
```

Special methods for any class

Method	Description
__new__(*cls*[, ...])	Instance creation (on construction). If __new__ returns an instance of *cls* then __init__ is called with the rest of the arguments (...), otherwise __init__ is not invoked. More details here.
__init__(*self*, *args*)	Instance initialization (on construction)
__del__(*self*)	Called on object demise (refcount becomes 0)
__repr__(*self*)	`repr()` and `` `...` `` conversions
__str__(*self*)	`str()` and `print` statement
__sizeof__(*self*)	Returns amount of memory used by object, in bytes (called by `sys.getsizeof()`).
__format__(*self, format_spec*)	`format()` and `str.format()` conversions
__cmp__(*self*,*other*)	Compares *self* to *other* and returns <0, 0, or >0. Implements >, <, == etc...
__index__(*self*)	[PEP357] Allows using any object as integer indice (e.g. for slicing). Must return a single integer or long integer value.
__lt__(*self*, *other*)	Called for *self* < *other* comparisons. Can return anything, or can raise an exception.
__le__(*self*, *other*)	Called for *self* <= *other* comparisons. Can return anything, or can raise an exception.
__gt__(*self*, *other*)	Called for *self* > *other* comparisons. Can return anything, or can raise an exception.
__ge__(*self*, *other*)	Called for *self* >= *other* comparisons. Can return anything, or can raise an exception.
__eq__(*self*, *other*)	Called for *self* == *other* comparisons. Can return anything, or can raise an exception.
__ne__(*self*, *other*)	Called for *self* != *other* (and *self* <> *other*) comparisons. Can return anything, or can raise an exception.
__hash__(*self*)	Compute a 32 bit hash code; `hash()` and dictionary ops. Since 2.5 can also return a **long** integer, in which case the hash of that value will be taken.Since 2.6 can set `__hash__ = None` to void class inherited hashability.
__nonzero__(*self*)	Returns 0 or 1 for truth value testing. when this method is not defined, `__len__()` is called if defined; otherwise all class instances are considered "true".
__getattr__(*self*,*name*)	Called when attribute lookup doesn't find *name*. See also __getattribute__.
__getattribute__(*self*, *name*)	Same as __getattr__ but **always** called whenever the attribute *name* is accessed.
__dir__(*self*)	Returns the list of names of valid attributes for the object. Called by builtin fct `dir()`, but ignored unless `__getattr__` or `__getattribute__` is defined.
__setattr__(*self*, *name*, *value*)	Called when setting an attribute (inside, don't use "*self.name* = *value*", use instead "*self.*__dict__[*name*] = *value*")
__delattr__(*self*, *name*)	Called to delete attribute <*name*>.
__call__(*self*, **args*, ***kwargs*)	Called when an instance is called as function: `obj(arg1, arg2, ...)` is a shorthand for `obj.__call__(arg1, arg2, ...)`.

Operators

See list in the `operator` module. Operator function names are provided with **2 variants**, with or without leading & trailing '__' (e.g. `__add__` or `add`).

Numeric operations special methods	
Operator	**Special method**
self **+** *other*	**__add__**(*self*, *other*)
self - *other*	**__sub__**(*self*, *other*)
self ***** *other*	**__mul__**(*self*, *other*)
self **/** *other*	**__div__**(*self*, *other*) or **__truediv__**(*self*,*other*) if `__future__.division` is active.
self **//** *other*	**__floordiv__**(*self*, *other*)
self **%** *other*	**__mod__**(*self*, *other*)
divmod(*self*,*other*)	**__divmod__**(*self*, *other*)
self ****** *other*	**__pow__**(*self*, *other*)
self **&** *other*	**__and__**(*self*, *other*)
self **^** *other*	**__xor__**(*self*, *other*)
self **|** *other*	**__or__**(*self*, *other*)
self **<<** *other*	**__lshift__**(*self*, *other*)
self **>>** *other*	**__rshift__**(*self*, *other*)
bool(*self*)	**__nonzero__**(*self*) (used in boolean testing)
-*self*	**__neg__**(*self*)
+*self*	**__pos__**(*self*)
abs(*self*)	**__abs__**(*self*)
~*self*	**__invert__**(*self*) (bitwise)
self **+=** *other*	**__iadd__**(*self*, *other*)
self **-=** *other*	**__isub__**(*self*, *other*)
self ***=** *other*	**__imul__**(*self*, *other*)
self **/=** *other*	**__idiv__**(*self*, *other*) or **__itruediv__**(*self*,*other*) if `__future__.division` is in effect.
self **//=** *other*	**__ifloordiv__**(*self*, *other*)
self **%=** *other*	**__imod__**(*self*, *other*)
self ****=** *other*	**__ipow__**(*self*, *other*)
self **&=** *other*	**__iand__**(*self*, *other*)
self **^=** *other*	**__ixor__**(*self*, *other*)
self **|=** *other*	**__ior__**(*self*, *other*)
self **<<=** *other*	**__ilshift__**(*self*, *other*)
self **>>=** *other*	**__irshift__**(*self*, *other*)

Conversions	
built-in function	**Special method**
int(*self*)	**__int__**(*self*)
long(*self*)	**__long__**(*self*)
float(*self*)	**__float__**(*self*)
complex(*self*)	**__complex__**(*self*)
oct(*self*)	**__oct__**(*self*)
hex(*self*)	**__hex__**(*self*)
coerce(*self*, *other*)	**__coerce__**(*self*, *other*)

Right-hand-side equivalents for all **binary** operators exist (__**r**add__, __**r**sub__, __**r**mul__, __**r**div__, ...). They are called when class instance is on r-h-s of operator:

- a + 3 calls `__add__(a, 3)`
- 3 + a calls `__radd__(a, 3)`

Special operations for *containers*		
Operation	**Special method**	**Notes**
All sequences and maps :		
len(*self*)	**__len__**(*self*)	length of object, >= 0. Length 0 == false
*self***[***k***]**	**__getitem__**(*self*, *k*)	Get element at indice /key k (indice starts at 0). Or, if *k* is a slice object, return a slice.
	__missing__(*self*, *key*)	Hook called when *key* is not found in the dictionary,

		returns the default value.
self[k] **=** *value*	**__setitem__**(*self*, *k*, *value*)	Set element at indice/key/slice *k*.
del *self*[*k*]	**__delitem__**(*self*, *k*)	Delete element at indice/key/slice *k*.
elt **in** *self* *elt* **not in** *self*	**__contains__**(*self*, *elt*) **not __contains__**(*self*, *elt*)	More efficient than std iteration thru sequence.
iter(*self*)	**__iter__**(*self*)	Returns an iterator on elements (keys for mappings <=> *self*.**iterkeys**()). See iterators.
Sequences, general methods, plus:		
self[*i*:*j*]	**__getslice__**(*self*, *i*, *j*)	**Deprecated** since 2.0, replaced by `__getitem__` with a slice object as parameter.
self[i:j] **=** *seq*	**__setslice__**(*self*, *i*, *j*,*seq*)	**Deprecated** since 2.0, replaced by `__setitem__` with a slice object as parameter.
del *self*[i:j]	**__delslice__**(*self*, *i*, *j*)	Same as self[i:j] = [] - **Deprecated** since 2.0, replaced by `__delitem__` with a slice object as parameter.
self ***** *n*	**__mul__**(*self*, *n*)	(**__repeat__** in the official doc but doesn't work!)
self **+** *other*	**__add__**(*self*, *other*)	(**__concat__** in the official doc but doesn't work!)
Mappings, general methods, plus:		
hash(*self*)	**__hash__**(*self*)	hashed value of object *self* is used for dictionary keys

Special informative state attributes for some types:

Tip: use module inspect to inspect live objects.

Lists & Dictionaries

Attribute	Meaning
~~**__methods__**~~	~~(list, R/O): list of method names of the object~~ **Deprecated**, use `dir()` instead

Modules

Attribute	Meaning
__doc__	(string/None, R/O): doc string (<=> __dict__['__doc__'])
__name__	(string, R/O): module name (also in __dict__['__name__'])
__package__	(string/None, R/W): If defined, package name used for relative imports (also in __dict__ ['__package__']). [PEP366].
__dict__	(dict, R/O): module's name space
__file__	(string/undefined, R/O): pathname of .pyc, .pyo or .pyd (undef for modules statically linked to the interpreter). Before 2.3 use `sys.argv[0]` instead to find the current script filename.
__path__	(list/undefined, R/W): List of directory paths where to find the package (for packages only).

Classes

Attribute	Meaning
__doc__	(string/None, R/W): doc string (<=> __dict__['__doc__'])
__name__	(string, R/**W**): class name (also in __dict__['__name__'])
__module__	(string, R/W): module name in which the class was defined
__bases__	(tuple, R/**W)**: parent classes
__dict__	(dict, R/**W**): attributes (class name space)

Instances

Attribute	Meaning
__class__	(class, R/W): instance's class
__dict__	(dict, R/W): attributes

User defined functions

Attribute	Meaning
__doc__	(string/None, R/**W**): doc string
__name__	(string, R/O): function name
func_doc	(R/**W**): same as __doc__
func_name	(R/O, R/W from 2.4): same as __name__
func_defaults	(tuple/None, R/**W**): default args values if any

func_code	(code, R/**W**): code object representing the compiled function body
func_globals	(dict, R/O): ref to dictionary of func global variables

User-defined Methods

Attribute	Meaning
__doc__	(string/None, R/O): Doc string
__name__	(string, R/O): Method name (same as im_func.__name__)
im_class	(class, R/O): Class defining the method (may be a base class)
im_self	(instance/None, R/O): Target instance object (None if unbound). Since 2.6 use `__self__` instead, will be deprecated in 3.0.
__self__	(instance/None, R/O): Target instance object (None if unbound).
im_func	(function, R/O): Function object. Since 2.6 use `__func__` instead, will be deprecated in 3.0.
__func__	(function, R/O): Function object.

Built-in Functions & methods

Attribute	Meaning
__doc__	(string/None, R/O): doc string
__name__	(string, R/O): function name
__self__	[methods only] target object
~~**__members__**~~	~~list of attr names: ['__doc__','__name__','__self__'])~~ **Deprecated**, use `dir()` instead.

Codes

Attribute	Meaning
co_name	(string, R/O): function name
co_argcount	(int, R/0): number of positional args
co_nlocals	(int, R/O): number of local vars (including args)
co_varnames	(tuple, R/O): names of local vars (starting with args)
co_code	(string, R/O): sequence of bytecode instructions
co_consts	(tuple, R/O): literals used by the bytecode, 1st one is function doc (or None)
co_names	(tuple, R/O): names used by the bytecode
co_filename	(string, R/O): filename from which the code was compiled
co_firstlineno	(int, R/O): first line number of the function
co_lnotab	(string, R/O): string encoding bytecode offsets to line numbers.
co_stacksize	(int, R/O): required stack size (including local vars)
co_flags	(int, R/O): flags for the interpreter bit 2 set if fct uses "*arg" syntax, bit 3 set if fct uses '**keywords' syntax

Frames

Attribute	Meaning
f_back	(frame/None, R/O): previous stack frame (toward the caller)
f_code	(code, R/O): code object being executed in this frame
f_locals	(dict, R/O): local vars
f_globals	(dict, R/O): global vars
f_builtins	(dict, R/O): built-in (intrinsic) names
f_restricted	(int, R/O): flag indicating whether fct is executed in restricted mode
f_lineno	(int, R/O): current line number
f_lasti	(int, R/O): precise instruction (index into bytecode)
f_trace	(function/None, R/W): debug hook called at start of each source line
f_exc_type	(Type/None, R/W): Most recent exception type
f_exc_value	(any, R/W): Most recent exception value
f_exc_traceback	(traceback/None, R/W): Most recent exception traceback

Tracebacks

Attribute	Meaning
tb_next	(frame/None, R/O): next level in stack trace (toward the frame where the exception occurred)
tb_frame	(frame, R/O): execution frame of the current level
tb_lineno	(int, R/O): line number where the exception occured
tb_lasti	(int, R/O): precise instruction (index into bytecode)

Slices

Attribute	Meaning
start	(any/None, R/O): lowerbound, included
stop	(any/None, R/O): upperbound, excluded

step	(any/None, R/O): step value

Complex numbers	
Attribute	**Meaning**
real	(float, R/O): real part
imag	(float, R/O): imaginary part

xranges	
Attribute	**Meaning**
tolist	(Built-in method, R/O): ?

Important Modules

sys

System-specific parameters and functions.

Some *sys* variables	
Variable	**Content**
argv	The list of command line arguments passed to a Python script. `sys.argv[0]` is the script name.
builtin_module_names	A list of strings giving the names of all modules written in C that are linked into this interpreter.
byteorder	Native byte order, either 'big'(-endian) or 'little'(-endian).
~~check_interval~~	How often to check for thread switches or signals (measured in number of virtual machine instructions). Deprecated (since ??), use method `getcheckinterval()` instead.
copyright	A string containing the copyright pertaining to the Python interpreter.
dont_write_bytecode	If `True`, prevents Python from from writing .pyc or .pyo files (same as invocation option -B).
exec_prefix prefix	Root directory where platform-dependent Python files are installed, e.g. 'C:\\Python23', '/usr'.
executable	Name of executable binary of the Python interpreter (e.g. 'C:\\Python23 \\python.exe', '/usr/bin/python')
exitfunc	User can set to a parameterless function. It will get called before interpreter exits. Deprecated since 2.4. Code should be using the existing `atexit` module
flags	Status of command line flags, as a R/O struct. [details]
float_info	A `structseq` holding information about the float type (precision, internal representation, etc...). [details]
last_type, last_value, last_traceback	Set only when an exception not handled and interpreter prints an error. Used by debuggers.
maxint	Maximum positive value for integers. Since 2.2 integers and long integers are unified, thus integers have no limit.
maxunicode	Largest supported code point for a Unicode character.
modules	Dictionary of modules that have already been loaded.
path	Search path for external modules. Can be modified by program. `sys.path[0]` == directory of script currently executed.
platform	The current platform, e.g. "sunos5", "win32"
ps1, ps2	Prompts to use in interactive mode, normally ">>>" and "..."
stdin, stdout, stderr	File objects used for I/O. One can redirect by assigning a new file object to them (or **any** object: with a method `write(`*`string`*`)` for stdout/stderr, or with a method `readline()` for stdin). `__stdin__`,`__stdout__` and `__stderr__` are the default values.
subversion	Info about Python build version in the Subversion repository: tuple (interpreter-name, branch-name, revision-range), e.g. `('CPython', 'tags/r25', '51908')`.
version	String containing version info about Python interpreter.
version_info	Tuple containing Python version info - *(major, minor, micro, level, serial)*.
winver	Version number used to form registry keys on Windows platforms (e.g. '2.2').

Some *sys* functions

Function	Result
_current_frames()	Returns the current stack frames for all running threads, as a dictionary mapping thread identifiers to the topmost stack frame currently active in that thread at the time the function is called.
displayhook	The function used to display the output of commands issued in interactive mode - defaults to the builtin `repr()`. `__displayhook__` is the original value.
excepthook	Can be set to a user defined function, to which any uncaught exceptions are passed. `__excepthook__` is the original value.
exit(*n*)	Exits with status *n* (usually 0 means OK). Raises `SystemExit` exception (hence can be caught and ignored by program)
getcheckinterval() / setcheckinterval(*interval*)	Gets / Sets the interpreter's thread switching interval (in number of bytecode instructions, default: 10 until 2.2, 100 from 2.3).
getrefcount(*object*)	Returns the reference count of the object. Generally 1 higher than you might expect, because of *object* arg temp reference.
getsizeof(*object*[, *default*])	Returns the amount of memory used by *object*, in bytes. Calls `o.__sizeof__()` if available. *default* returned if size can't be determined. [details]
settrace(*func*)	Sets a trace function: called before each line of code is exited.
setprofile(*func*)	Sets a profile function for performance profiling.
exc_info()	Info on exception currently being handled; this is a tuple (exc_type, exc_value, exc_traceback). **Warning**: assigning the traceback return value to a local variable in a function handling an exception will cause a circular reference.
setdefaultencoding (*encoding*)	Change default Unicode encoding - defaults to 7-bit ASCII.
getrecursionlimit()	Retrieve maximum recursion depth.
setrecursionlimit()	Set maximum recursion depth (default 1000).

os

Miscellaneous operating system interfaces. **Many** functions, see the for a comprehensive list!

"synonym" for whatever OS-specific module (nt, mac, posix...) is proper for current environment. This module uses posix whenever possible.
See also M.A. Lemburg's utility `platform.py` (now included in 2.3+).

Some *os* variables

Variable	Meaning
name	name of O/S-specific module (e.g. "posix", "mac", "nt")
path	O/S-specific module for path manipulations. On Unix, `os.path.split()` <=> `posixpath.split()`
curdir	string used to represent current directory (eg '.')
pardir	string used to represent parent directory (eg '..')
sep	string used to separate directories ('/' or '\'). **Tip**: Use `os.path.join()` to build portable paths.
altsep	Alternate separator if applicable (`None` otherwise)
pathsep	character used to separate search path components (as in $PATH), eg. ';' for windows.
linesep	line separator as used in **text** files, ie '\n' on Unix, '\r\n' on Dos/Win, '\r' on Mac.

Some *os* functions

Function	Result
makedirs(*path*[, *mode*=0777])	Recursive directory creation (create required intermediary dirs); `os.error` if fails.
removedirs(*path*)	Recursive directory delete (delete intermediary **empty** dirs); fails (`os.error)` if the directories are not empty.
renames(*old*, *new*)	Recursive directory or file renaming; `os.error` if fails.
urandom(*n*)	Returns a string containing *n* bytes of random data.

posix

Posix OS interfaces.
Do **not** import this module directly, import os instead ! (see also module: shutil for file copy & remove functions)

***posix* Variables**

Variable	Meaning
environ	dictionary of environment variables, e.g. `posix.environ['HOME']`.
error	exception raised on POSIX-related error. Corresponding value is tuple of errno code and *perror()* string.

Some *posix* functions

Function	Result
access(*path*, *mode*)	Returns True if the requested access to *path* is granted. Use *mode*=`F_OK` to check for existence, or an OR-ed combination of `R_OK`, `W_OK`, and `X_OK` to check for r, w, x permissions.
chdir(*path*)	Changes current directory to *path*.
chmod(*path*, *mode*)	Changes the mode of *path* to the numeric *mode*
close(*fd*)	Closes file descriptor *fd* opened with posix.open.
_exit(*n*)	Immediate exit, with no cleanups, no SystemExit, etc... Should use this to exit a child process.
execv(*p*, *args*)	"Become" executable *p* with args *args*
getcwd()	Returns a string representing the current working directory.
getcwdu()	Returns a **Unicode** string representing the current working directory.
getpid()	Returns the current process id.
getsid()	Calls the system call getsid() [Unix].
fork()	Like C's fork(). Returns 0 to child, child pid to parent [Not on Windows].
kill(*pid*, *signal*)	Like C's kill [Not on Windows].
listdir(*path*)	Lists (base)names of entries in directory *path*, excluding '.' and '..'. If *path* is a Unicode string, so will be the returned strings.
lseek(*fd*, *pos*, *how*)	Sets current position in file *fd* to position *pos*, expressed as an offset relative to beginning of file (*how*=0), to current position (*how*=1), or to end of file (*how*=2).
mkdir(*path*[, *mode*])	Creates a directory named *path* with numeric *mode* (default 0777). Actual permissions = (*mode* & ~umask & 0777). To set **directly** the permissions, use `chmod()` after dir creation.
open(*file*, *flags*, *mode*)	Like C's open(). Returns file descriptor. Use file object functions rather than this low level ones.
pipe()	Creates a pipe. Returns pair of file descriptors (r, w) [Not on Windows].
popen(*command*, *mode*='r', *bufSize*=0)	Opens a pipe to or from *command*. Result is a file object to read to or write from, as indicated by *mode* being 'r' or 'w'. Use it to catch a command output ('r' mode), or to feed it ('w' mode).
remove(*path*)	See `unlink`.
rename(*old*, *new*)	Renames/moves the file or directory *old* to *new*. [error if target name already exists]
renames(*old*, *new*)	Recursive directory or file renaming function. Works like rename(), except **creation** of any intermediate directories needed to make the new pathname good is attempted first. After the rename, directories corresponding to rightmost path segments of the old name will be **pruned** away using removedirs().
rmdir(*path*)	Removes the empty directory *path*
read(*fd*, *n*)	Reads *n* bytes from file descriptor *fd* and return as string.
stat(*path*)	Returns st_mode, st_ino, st_dev, st_nlink, st_uid,st_gid, st_size, st_atime, st_mtime, st_ctime. [st_ino, st_uid, st_gid are dummy on Windows]
system(*command*)	Executes string *command* in a subshell. Returns exit status of subshell (usually 0 means OK). Since 2.4 use subprocess.call() instead.
times()	Returns accumulated CPU times in sec (user, system, children's user, children's sys, elapsed real time) [3 last not on Windows].
unlink(*path*)	Unlinks ("deletes") the file (not dir!) *path*. Same as: `remove`.
utime(*path*, (*aTime*, *mTime*))	Sets the access & modified time of the file to the given tuple of values.
wait()	Waits for child process completion. Returns tuple of pid, exit_status [Not on Windows].
waitpid(*pid*, *options*)	Waits for process *pid* to complete. Returns tuple of *pid*, exit_status [Not on Windows].
walk(*top*[, *topdown*=True [, *onerror*=None[, *followlinks*=False]]])	Generates a list of file names in a directory tree, by walking the tree either top down or bottom up. For each directory in the tree rooted at directory *top* (including top itself), it yields a 3-tuple (dirpath, dirnames, filenames) - more info here. See also `os.path.walk()`. 2.6: New *followlinks* parameter. If `True`, visit directories pointed to by links

	(beware of infinite recursion!).
write(*fd*, *str*)	Writes *str* to file *fd*. Returns nb of bytes written.

posixpath

Posix pathname operations.
Do **not** import this module directly, import os instead and refer to this module as **os.path**. (e.g. `os.path.exists(p)`)!

***posixpath* functions**

Function	Result
abspath(*path*)	Returns absolute path for *path*, taking current working dir in account.
commonprefix(*list*)	Returns the longuest path prefix (taken character-by-character) that is a prefix of all paths in list (or '' if *list* empty).
dirname/basename (*path*)	directory and name parts of *path*. See also split.
exists(*path*)	True if *path* is the path of an existing file or directory. See also `lexists`.
expanduser(*path*)	Returns a copy of *path* with "~" expansion done.
expandvars(*path*)	Returns string that is (a copy of) *path* with environment vars `$name` or `${name}` expanded. [Windows: case significant; ~~must use Unix: $var notation, not %var%~~; 2.6: Notation `%name%` **also** supported.]
getatime(*path*)	Returns last access time of *path* (integer nb of seconds since epoch).
getctime(*path*)	Returns the metadata change time of *path* (integer nb of seconds since epoch).
getmtime(*path*)	Returns last modification time of *path* (integer nb of seconds since epoch).
getsize(*path*)	Returns the size in bytes of *path*. `os.error` if file inexistent or inaccessible.
isabs(*path*)	True if *path* is absolute.
isdir(*path*)	True if *path* is a directory.
isfile(*path*)	True if *path* is a *regular* file.
islink(*path*)	True if *path* is a symbolic link.
ismount(*path*)	True if *path* is a mount point [true for all dirs on Windows].
join(*p*[,*q*[,...]])	Joins one or more path components in a way suitable for the current OS.
lexists(*path*)	True if the file specified by *path* exists, whether or not it's a symbolic link (unlike `exists`).
normcase(*path*)	Normalizes case of *path*. Has no effect under Posix.
normpath(*path*)	Normalizes *path*, eliminating double slashes, etc...
realpath(*path*)	Returns the canonical path for *path*, eliminating any symbolic links encountered in the path.
relpath(*path*[, *start*])	Returns a relative filepath to *path*, from the current directory by default, or from *start* if specified.
samefile(*f1*, *f2*)	True if the 2 paths *f1* and *f2* reference the same file.
sameopenfile(*f1*, *f2*)	True if the 2 open file objects *f1* and *f2* reference the same file.
samestat(*s1*, *s2*)	True if the 2 stat buffers *s1* and *s2* reference the same file.
split(*p*)	Splits *p* into (head, tail) where *tail* is last pathname component and *head* is everything leading up to that. <=> `(dirname(p), basename(p))`
splitdrive(*p*)	Splits path *p* in a pair ('drive:', tail) [Windows]
splitext(*p*)	Splits into (root, ext) where last comp of *root* contains no periods and *ext* is empty or starts with a period. 2.6: Do not split on leading period.
walk(*p*, *visit*, *arg*)	Calls the function *visit* with arguments (*arg*, *dirname*, *names*) for each directory recursively in the directory tree rooted at *p* (including *p* itself if it's a dir). The argument *dirname* specifies the visited directory, the argument *names* lists the files in the directory. The *visit* function may modify *names* to influence the set of directories visited below *dirname*, e.g. to avoid visiting certain parts of the tree. **See also** `os.walk()` for an alternative.

shutil

High-level file operations (copying, deleting).

Main *shutil* functions

Function	Result
copy(*src*, *dest*)	Copies the contents of file *src* to file *dest*, retaining file permissions.
copytree(*src*, *dest*[, *symlinks*=False[, *ignore*=None]])	Recursively copies an entire directory tree rooted at *src* into *dest* (which should not already exist). If *symlinks* is true, links in *src* are kept as such in *dest*. 2.6: New *ignore* callable argument. Will be called with each directory path and a list

	of the directory's contents, must return a list of names to ignore. `shutil.ignore_patterns()` can be used to exclude glob-style patterns, e.g.: `shutil.copytree('projects/myProjUnderSvn', 'exportDir', ignore=shutil.ignore_patterns('*~', '.svn'))`
move(*src*, *dest*)	Recursively moves a file or directory to a new location.
rmtree(*path*[, *ignore_errors*[, *onerror*]])	Deletes an entire directory tree, ignoring errors if *ignore_errors* is true, or calling *onerror*(func, path, sys.exc_info()) if supplied, with arguments *func* (faulty function), and *path* (concerned file). This fct fails when the files are Read Only.

(and also: *copyfile, copymode, copystat, copy2*)

time

Time access and conversions.
(see also module mxDateTime if you need a more sophisticated date/time management)

Variables

Variable	Meaning
altzone	Signed offset of local DST timezone in sec west of the 0th meridian.
daylight	Non zero if a DST timezone is specified.
timezone	The offset of the local (non-DST) timezone, in seconds west of UTC.
tzname	A tuple (name of local non-DST timezone, name of local DST timezone).

Some functions

Function	Result
clock()	On Unix: current processor time as a floating point number expressed in seconds. On Windows: wall-clock seconds elapsed since the 1st call to this function, as a floating point number (precision < 1µs).
time()	Returns a float representing UTC time in **seconds** since the epoch.
gmtime([*secs*]), **localtime**([*secs*])	Returns a 9-tuple representing time. Current time is used if *secs* is not provided. Since 2.2, returns a `struct_time` object (still accessible as a tuple) with the following attributes: (see table below)
asctime([*timeTuple*]),	24-character string of the following form: 'Mon Apr 03 08:31:14 2006'. *timeTuple* defaults to `localtime()` if omitted.
ctime([*secs*])	equivalent to `asctime(localtime(`*secs*`))`
mktime(*timeTuple*)	Inverse of `localtime`(). Returns a float representing a number of seconds.
strftime(*format*[, *timeTuple*])	**Formats** a time tuple as a string, according to *format* (see table below). Current time is used if *timeTuple* is omitted.
strptime(*string*[, *format*])	**Parses** a string representing a time according to *format* (same format as for `strftime()`, see below), default "%a %b %d %H:%M:%S %Y" = asctime format. Returns a time tuple/`struct_time`.
sleep(*secs*)	Suspends execution for *secs* seconds. *secs* can be a float.

Index	Attribute	Values
0	tm_year	Year (e.g. 1993)
1	tm_mon	Month [1,12]
2	tm_mday	Day [1,31]
3	tm_hour	Hour [0,23]
4	tm_min	Minute [0,59]
5	tm_sec	Second [0,**61**]; The 61 accounts for leap seconds and (the very rare) double leap seconds.
6	tm_wday	Weekday [0,6], Monday is 0
7	tm_yday	Julian day [1,366]
8	tm_isdst	Daylight flag: 0, 1 or -1; -1 passed to `mktime()` will usually work

Formatting in strftime() and strptime()

Directive	Meaning
`%a`	Locale's abbreviated weekday name.
`%A`	Locale's full weekday name.

`%b`	Locale's abbreviated month name.
`%B`	Locale's full month name.
`%c`	Locale's appropriate date and time representation.
`%d`	Day of the month as a decimal number [01,31].
`%H`	Hour (24-hour clock) as a decimal number [00,23].
`%I`	Hour (12-hour clock) as a decimal number [01,12].
`%j`	Day of the year as a decimal number [001,366].
`%m`	Month as a decimal number [01,12].
`%M`	Minute as a decimal number [00,59].
`%p`	Locale's equivalent of either AM or PM.
`%S`	Second as a decimal number [00,61]. Yes, 61 !
`%U`	Week number of the year (Sunday as the first day of the week) as a decimal number [00,53]. All days in a new year preceding the first Sunday are considered to be in week 0.
`%w`	Weekday as a decimal number [0(Sunday),6].
`%W`	Week number of the year (Monday as the first day of the week) as a decimal number [00,53]. All days in a new year preceding the first Sunday are considered to be in week 0.
`%x`	Locale's appropriate date representation.
`%X`	Locale's appropriate time representation.
`%y`	Year without century as a decimal number [00,99].
`%Y`	Year with century as a decimal number.
`%Z`	Time zone name (or by no characters if no time zone exists).
`%%`	A literal "`%`" character.

string

Common string operations.
As of Python 2.0, much (though not all) of the functionality provided by the string module have been superseded by built-in string methods - see Operations on strings for details.

Some *string* variables	
Variable	**Meaning**
digits	The string '0123456789'.
hexdigits, octdigits	Legal hexadecimal & octal digits.
letters, uppercase, lowercase, whitespace	Strings containing the appropriate characters.
ascii_letters, ascii_lowercase, ascii_uppercase	Same, taking the current *locale* in account.
index_error	Exception raised by `index()` if substring not found.

Some *string* functions	
Function	**Result**
expandtabs(*s*, *tabSize*)	Returns a copy of string *s* with tabs expanded.
find/rfind(*s*, *sub*[, *start*=0[, *end*=0])	Returns the lowest/highest index in *s* where the substring *sub* is found such that *sub* is wholly contained in *s*[*start*:*end*]. Return -1 if *sub* not found.
ljust/rjust/center(*s*, *width*[, *fillChar*=' '])	Returns a copy of string *s*; left/right justified/centered in a field of given width, padded with spaces or the given character. *s* is never truncated.
lower/upper(*s*)	Returns a string that is (a copy of) *s* in lowercase/uppercase.
split(*s*[, *sep*=whitespace[, *maxsplit*=0]])	Returns a list containing the words of the string *s*, using the string *sep* as a separator.
rsplit(*s*[, *sep*=whitespace[, *maxsplit*=0]])	Same as `split` above but starts splitting from the end of string, e.g. `'A,B,C'.split(',', 1) == ['A', 'B,C']` but `'A,B,C'.rsplit(',', 1) == ['A,B', 'C']`
join(*words*[, *sep*=' '])	Concatenates a list or tuple of words with intervening separators; inverse of `split`.
replace(*s*, *old*, *new*[, *maxsplit*=0]	Returns a copy of string *s* with all occurrences of substring *old* replaced by *new*. Limits to *maxsplit* first substitutions if specified.
strip(*s*[, *chars*=None])	Returns a string that is (a copy of) *s* without leading and trailing *chars* (default: whitespace), if any. Also: `lstrip`, `rstrip`.

re (sre)

Regular expression operations.

Handles Unicode strings. Implemented in new module **sre**, **re** now a mere front-end for compatibility.
Patterns are specified as strings. Tip: Use **raw** strings (e.g. `r'\w*'`) to literalize backslashes.

Regular expression syntax

Form	Description
.	Matches any character (including newline if DOTALL flag specified).
^	Matches start of the string (of every line in MULTILINE mode).
$	Matches end of the string (of every line in MULTILINE mode).
*	0 or more of preceding regular expression (as **many** as possible).
+	1 or more of preceding regular expression (as **many** as possible).
?	0 or 1 occurrence of preceding regular expression.
*?, +?, ??	Same as *, + and ? but matches as **few** characters as possible.
{m,n}	Matches from m to n repetitions of preceding RE.
{m,n}?	Idem, attempting to match as **few** repetitions as possible.
[ ]	Defines character set: e.g. '[a-zA-Z]' to match all letters (see also \w \S).
[^]	Defines complemented character set: matches if char is NOT in set.
\	Escapes special chars '*?+&$|()' and introduces special sequences (see below). Due to Python string rules, write as '\\' or r'\' in the pattern string.
\\	Matches a litteral '\'; due to Python string rules, write as '\\\\' in pattern string, or better using raw string: r'\\'.
|	Specifies alternative: 'foo|bar' matches 'foo' or 'bar'.
(...)	Matches any RE inside (), and delimits a *group*.
(?:...)	Idem but doesn't delimit a *group* (*non capturing* parenthesis).
(?P<*name*>...)	Matches any RE inside (), and delimits a **named** *group*, (e.g. r'(?P<*id*>[a-zA-Z_]\w*)' defines a group named *id*).
(?P=*name*)	Matches whatever text was matched by the earlier group named *name*.
(?=...)	Matches if ... matches next, but doesn't consume any of the string e.g. 'Isaac (?=Asimov)' matches 'Isaac' only if followed by 'Asimov'.
(?!...)	Matches if ... **doesn't** match next. Negative of (?=...).
(?<=...)	Matches if the current position in the string is preceded by a match for ... that ends at the current position. This is called a *positive lookbehind assertion*.
(?<!...)	Matches if the current position in the string is not preceded by a match for .... This is called a *negative lookbehind assertion*.
(?(*group*) *A*|*B*)	[2.4+] *group* is either a numeric group ID or a group name defined with `(?Pgroup...)` earlier in the expression. If the specified group matched, the regular expression pattern *A* will be tested against the string; if the group didn't match, the pattern *B* will be used instead.
(?#...)	A comment; ignored.
(?*letters*)	*letters* is one or more of 'i','L', 'm', 's', 'u', 'x'. Sets the corresponding **flags** (re.I, re.L, re.M, re.S, re.U, re.X) for the entire RE. See the `compile()` function for equivalent flags.

Special sequences

Sequence	Description
\number	Matches content of the *group* of the same number; groups are numbered starting from 1.
\A	Matches only at the start of the string.
\b	Empty str at beginning or end of *word*: '\bis\b' matches 'is', but not 'his'.
\B	Empty str NOT at beginning or end of word.
\d	Any decimal digit (<=> [0-9]).
\D	Any non-decimal digit char (<=> [^0-9]).
\s	Any whitespace char (<=> [\t\n\r\f\v]).
\S	Any non-whitespace char (<=> [^ \t\n\r\f\v]).
\w	Any alphaNumeric char (depends on LOCALE flag).
\W	Any non-alphaNumeric char (depends on LOCALE flag).
\Z	Matches only at the end of the string.

Variables

Variable	Meaning
error	Exception when pattern string isn't a valid regexp.

Functions

Function	Result
compile(*pattern*[,*flags*=0])	Compiles a RE pattern string into a *regular expression object*. Flags (combinable by |): *I or IGNORECASE <=> (?i)* case insensitive matching *L or LOCALE <=> (?L)* make \w, \W, \b, \B dependent on the current locale *M or MULTILINE <=> (?m)* matches every new line and not only start/end of the whole string *S or DOTALL <=> (?s)* '.' matches ALL chars, including newline *U or UNICODE <=> (?u)* Make \w, \W, \b, and \B dependent on the Unicode character properties database. *X or VERBOSE <=> (?x)* Ignores whitespace outside character sets
escape(*string*)	Returns (a copy of) *string* with all non-alphanumerics backslashed.
match(*pattern*, *string*[, *flags*])	If 0 or more chars at **beginning** of *string* matches the RE pattern string, returns a corresponding *MatchObject* instance, or `None` if no match.
search(*pattern*, *string*[, *flags*])	Scans thru *string* for a location matching *pattern*, returns a corresponding *MatchObject* instance, or `None` if no match.
split(*pattern*, *string*[, *maxsplit*=0])	Splits *string* by occurrences of *pattern*. If capturing () are used in pattern, then occurrences of patterns or subpatterns are also returned.
findall(*pattern*, *string*)	Returns a list of non-overlapping matches of *pattern* in *string*, either a list of groups or a list of tuples if the pattern has more than 1 group.
finditer(*pattern*, *string*[, *flags*])	Returns an iterator over all non-overlapping matches of *pattern* in *string*. For each match, the iterator returns a `match` object. Empty matches are included in the result unless they touch the beginning of another match.
sub(*pattern*, *repl*, *string*[, *count*=0])	Returns string obtained by replacing the (*count* first) leftmost non-overlapping occurrences of *pattern* (a string or a RE object) in *string* by *repl*; *repl* can be a string or a function called with a single *MatchObj* arg, which must return the replacement string.
subn(*pattern*, *repl*, *string*[, *count*=0])	Same as `sub()`, but returns a tuple (newString, numberOfSubsMade).

Regular Expression Objects

RE objects are returned by the compile function.

re object attributes

Attribute	Description
flags	Flags arg used when RE obj was compiled, or 0 if none provided.
groupindex	Dictionary of {group name: group number} in pattern.
pattern	Pattern string from which RE obj was compiled.

re object methods

Method	Result
match(*string*[, *pos*][, *endpos*])	If zero or more characters at the beginning of string match this regular expression, returns a corresponding `MatchObject` instance. Returns `None` if the string does not match the pattern; note that this is different from a zero-length match. The optional second parameter *pos* gives an index in the string where the search is to start; it defaults to 0. This is not completely equivalent to slicing the string; the '' pattern character matches at the real beginning of the string and at positions just after a newline, but not necessarily at the index where the search is to start. The optional parameter *endpos* limits how far the string will be searched; it will be as if the string is endpos characters long, so only the characters from *pos* to *endpos* will be searched for a match.
search(*string*[, *pos*][, *endpos*])	Scans through string looking for a location where this regular expression

	produces a match, and returns a corresponding `MatchObject` instance. Returns `None` if no position in the string matches the pattern; note that this is different from finding a zero-length match at some point in the string. The optional *pos* and *endpos* parameters have the same meaning as for the `match()` method.
split(*string*[, *maxsplit*=0])	Identical to the `split()` function, using the compiled pattern.
findall(*string*[, *pos*[, *endpos*]])	Identical to the `findall()` function, using the compiled pattern.
finditer(*string*[, *pos*[, *endpos*]])	Identical to the `finditer()` function, using the compiled pattern.
sub(*repl*, *string*[, *count*=0])	Identical to the `sub()` function, using the compiled pattern.
subn(*repl*, *string*[, *count*=0])	Identical to the `subn()` function, using the compiled pattern.

Match Objects

Match objects are returned by the match & search functions.

Match object attributes

Attribute	Description
pos	Value of pos passed to search or match functions; index into string at which RE engine started search.
endpos	Value of endpos passed to search or match functions; index into string beyond which RE engine won't go.
re	RE object whose match or search fct produced this `MatchObj` instance.
string	String passed to `match()` or `search()`.

Match object methods

Method	Result
group([*g1*, *g2*, ...])	Returns one or more groups of the match. If **one** arg, result is a string; if multiple args, result is a tuple with one item per arg. If *gi* is 0, returns the entire matching string; if 1 <= *gi* <= 99, returns string matching group #*gi* (or None if no such group); *gi* may also be a group *name*.
groups()	Returns a tuple of all groups of the match; groups not participating to the match have a value of None. Returns a string instead of tuple if len(tuple)== 1.
start(*group*), end(*group*)	Returns indices of start & end of substring matched by group (or None if group exists but didn't contribute to the match).
span(*group*)	Returns the 2-tuple (start(group), end(group)); can be (None, None) if group didn't contibute to the match.

math

For complex number functions, see module cmath. For intensive number crunching, see Numerical Python and the Python and Scientific computing page.

Constants

Name	Value
pi	3.1415926535897931
e	2.7182818284590451

Functions

Name	Result
acos(*x*)	Returns the arc cosine (measured in radians) of *x*.
acosh(*x*)	Returns the hyperbolic arc cosine (measured in radians) of *x*.
asin(*x*)	Returns the arc sine (measured in radians) of *x*.
asinh(*x*)	Returns the hyperbolic arc sine (measured in radians) of *x*.
atan(*x*)	Returns the arc tangent (measured in radians) of *x*.
atan2(*y*, *x*)	Returns the arc tangent (measured in radians) of *y*/*x*. The result is between -pi and pi. Unlike `atan(y/x)`, the signs of both *x* and *y* are considered.
atanh(*x*)	Returns the hyperbolic arc tangent (measured in radians) of *x*.
ceil(*x*)	Returns the ceiling of *x* as a float. This is the smallest integral value >= *x*.
copysign(*x*, *y*)	Copies the sign bit of an IEEE 754 number, returning the absolute value of *x* combined with the sign bit of *y*, e.g. `copysign(1, -0.0)` returns `-1.0`.
cos(*x*)	Returns the cosine of *x* (measured in radians).
cosh(*x*)	Returns the hyperbolic cosine of *x*.

degrees(*x*)	Converts angle *x* from radians to degrees.
exp(*x*)	Returns *e* raised to the power of *x*.
fabs(*x*)	Returns the absolute value of the float *x*.
factorial(*n*)	returns `n!`
floor(*x*)	Returns the floor of *x* as a float. This is the largest integral value <= *x*.
fmod(*x*, *y*)	Returns fmod(x, y), according to platform C. x % y may differ.
frexp(*x*)	Returns the mantissa and exponent of *x*, as pair `(m, e)`. m is a float and e is an int, such that x = m * 2.**e. If *x* is 0, m and e are both 0. Else 0.5 <= abs(m) < 1.0.
fsum(*iterable*)	Returns an **accurate** floating point sum of values in *iterable* (assumes IEEE-754 floating point arithmetic).
hypot(*x*, *y*)	Returns the Euclidean distance `sqrt(x*x + y*y)`.
isinf(*x*)	Returns True if *x* is infinite (positive or negative).
isnan(*x*)	Returns True if *x* is not a number.
ldexp(*x*, *i*)	x * (2**i)
log(*x*[, *base*])	Returns the logarithm of *x* to the given *base*. If the base is not specified, returns the natural logarithm (base e) of *x*.
log10(*x*)	Returns the base 10 logarithm of *x*.
log1p(*x*)	Returns the natural logarithm of 1+*x* (base e). The result is computed in a way which is accurate for *x* near zero.
modf(*x*)	Returns the fractional and integer parts of *x*. Both results carry the sign of *x*. The integer part is returned as a float.
pow(*x*, *y*)	Returns `x**y` (*x* to the power of *y*). Note that for *y*=2, it is more efficient to use `x*x`.
radians(*x*)	Converts angle *x* from degrees to radians.
sin(*x*)	Returns the sine (measured in radians) of *x*.
sinh(*x*)	Returns the hyperbolic sine of *x*.
sqrt(*x*)	Returns the square root of *x*.
tan(*x*)	Returns the tangent (measured in radians) of *x*.
tanh(*x*)	Returns the hyperbolic tangent of *x*.
trunc(*x*)	Returns the `Real` value *x* truncated to an `Integral`. Delegates to `x.__trunc__()`.

getopt

Parser for command line options.

This was the standard parser until Python 2.3, now superseded by optparse.
[see also: Richard Gruet's simple parser getargs.py (shameless self promotion)]

Functions:

```
getopt(list, optstr)    -- Similar to C. <optstr> is option letters to look for.
                           Put ':' after letter if option takes arg. E.g.
    # invocation was "python test.py -c hi -a arg1 arg2"
       opts, args =  getopt.getopt(sys.argv[1:], 'ab:c:')
    # opts would be
       [('-c', 'hi'), ('-a', '')]
    # args would be
       ['arg1', 'arg2']
```

List of modules and packages in base distribution

Built-ins and content of python **`Lib`** directory. The subdirectory `Lib/site-packages` contains platform-specific packages and modules.
[**Main distributions (Windows, Unix)**, some OS specific modules may be missing]

Standard library modules	
Operation	**Result**
__builtin__	Provide direct access to all `built-in' identifiers of Python, e.g. `__builtin__.open` is the full name for the built-in function `open()`.
__future__	Future statement definitions. Used to progressively introduce new features in the language.
__main__	Represent the (otherwise anonymous) scope in which the interpreter's main program executes -- commands read either from standard input, from a script

	file, or from an interactive prompt. Typical idiom to check if a code was run as a *script* (as opposed to being *imported*):
	`if __name__ == '__main__':`
	`    main()  # (this code was run as script)`

abc	(new in 2.6) **Abstract Base Classes (ABC)** [PEP 3119]. Equivalent of Java *interfaces*. The module collections defines interfaces/ABCs for many behaviors/protocols/data structures (`Iterable`, `Hashable`, `Sequence`, `Set`, etc...).
aifc	Stuff to parse AIFF-C and AIFF files.
anydbm	Generic interface to all dbm clones. (dbhash, gdbm, dbm, dumbdbm).
array	Efficient arrays of numeric values.
ast	(new in 2.6) Helpers to process Trees of the Python Abstract Syntax grammar.
asynchat	A class supporting chat-style (command/response) protocols.
asyncore	Basic infrastructure for asynchronous socket service clients and servers.
atexit	Register functions to be called at exit of Python interpreter.
audiodev	Classes for manipulating audio devices (currently only for Sun and SGI). Deprecated since 2.6.
audioop	Manipulate raw audio data. 2.5: Supports the a-LAW encoding.
base64	Conversions to/from base64 transport encoding as per RFC-1521.
BaseHTTPServer	HTTP server base class
~~Bastion~~	~~"Bastionification" utility (control access to instance vars).~~
bdb	A generic Python debugger base class.
binascii	Convert between binary and ASCII.
binhex	Macintosh binhex compression/decompression.
bisect	Bisection algorithms.
bsddb	(Optional) improved BSD database interface [package].
bz2	BZ2 compression.
calendar	Calendar printing functions.
cgi	Wraps the WWW Forms Common Gateway Interface (CGI).
CGIHTTPServer	CGI-savvy HTTP Server.
cgitb	Traceback manager for CGI scripts.
chunk	Read IFF chunked data.
cmath	Mathematical functions for complex numbers. See also math.
cmd	A generic class to build line-oriented command interpreters.
~~cmp~~	~~Efficiently compare files, boolean outcome only.~~
~~cmpcache~~	~~Same, but caches 'stat' results for speed.~~
code	Utilities needed to emulate Python's interactive interpreter.
codecs	Lookup existing Unicode encodings and register new ones. 2.5: support for incremental codecs.
codeop	Utilities to compile possibly incomplete Python source code.
collections	high-performance container datatypes. 2.4: The only datatype defined is a double-ended queue `deque`. 2.5: Type `deque` has now a `remove` method. New type `defaultdict`. 2.6: New type `namedtuple`. Define many ABCs (Abstract Base Classes) like Container, Hashable, Iterable, Sequence, Set...
colorsys	Conversion functions between RGB and other color systems.
commands	Execute shell commands via os.popen [Unix].
compileall	Force "compilation" of all .py files in a directory.
ConfigParser	Configuration file parser (much like windows .ini files).
contextlib	Utilities for `with` statement contexts.
Cookie	HTTP state (cookies) management.
copy	Generic shallow and deep copying operations.
copy_reg	Helper to provide extensibility for modules pickle/cPickle.
cPickle	Faster, C implementation of pickle.
cProfile	Faster, C implementation of profile.
crypt	Function to check Unix passwords [Unix].
cStringIO	Faster, C implementation of StringIO.
csv	Tools to read comma-separated files (of variations thereof). 2.5: Several enhancements.
ctypes	"Foreign function" library for Python. Provides C compatible data types, and allows to call functions in dlls/shared libraries. Can be used to wrap these libraries in pure Python.
curses	Terminal handling for character-cell displays [Unix/OS2/DOS only].
datetime	Improved date/time types (`date`, `time`, `datetime`, `timedelta`). 2.5: New method

	`strptime(string, format)` for class `datetime`. 2.6: `strftime()` new format code `%f` expanding to number of s.
dbhash	(g)dbm-compatible interface to bsdhash.hashopen.
decimal	Decimal floating point arithmetic.
difflib	Tool for comparing sequences, and computing the changes required to convert one into another. 2.5: Improved `SequenceMatcher.get_matching_blocks()` method .
dircache	Sorted list of files in a dir, using a cache. Deprecated since 2.6.
~~dircmp~~	~~Defines a class to build directory diff tools on.~~
dis	Bytecode disassembler.
distutils	Package installation system. 2.5: Function `setup` enhanced with new keyword parameters `requires`, `provides`, `obsoletes`, and `download_url` [PEP314].
distutils.command.register	Registers a module in the Python package index (PyPI). This command plugin adds the register command to distutil scripts.
distutils.debug	
distutils.emxccompiler	
distutils.log	
dl	Call C functions in shared objects [Unix]. Deprecated since 2.6.
doctest	Unit testing framework based on running examples embedded in docstrings. 2.5: New `SKIP` option. New *encoding* arg to `testfile()` function.
DocXMLRPCServer	Creation of self-documenting XML-RPC servers, using pydoc to create HTML API doc on the fly. 2.5: New attribute `rpc_paths`.
~~dospath~~	~~Common operations on DOS pathnames.~~
dumbdbm	A dumb and slow but simple dbm clone.
~~dump~~	~~Print python code that reconstructs a variable.~~
dummy_thread	
dummy_threading	Helpers to make it easier to write code that uses threads where supported, but still runs on Python versions without thread support. The dummy modules simply run the threads sequentially.
email	A package for parsing, handling, and generating email messages. New version 3.0 dropped various deprecated APIs and removes support for Python versions earlier than 2.3. 2.5: Updated to version 4.0.
encodings	New codecs: **idna** (IDNA strings), **koi8_u** (Ukranian), **palmos** (PalmOS 3.5), **punycode** (Punycode IDNA codec), **string_escape** (Python string escape codec: replaces non-printable chars w/ Python-style string escapes). New codecs in 2.4: HP Roman8, ISO_8859-11, ISO_8859-16, PCTP-154, TIS-620; Chinese, Japanese and Korean codecs.
errno	Standard errno system symbols. The value of each symbol is the corresponding integer value.
exceptions	Class based **built-in** exception hierarchy.
fcntl	The `fcntl()` and `ioctl()` system calls [Unix].
filecmp	File and directory comparison.
fileinput	Helper class to quickly write a loop over all standard input files. 2.5: Made more flexible (Unicode filenames, *mode* parameter, etc...)
~~find~~	~~Find files directory hierarchy matching a pattern.~~
fnmatch	Filename matching with shell patterns.
formatter	Generic output formatting.
fpectl	Floating point exception control [Unix].
fpformat	General floating point formatting functions. Deprecated since 2.6.
fractions	(new in 2.6) Rational Numbers.
ftplib	An FTP client class. Based on RFC 959.
functools	Tools for functional-style programming. See in particular function `partial()` [PEP309].

future_builtins	(new in 2.6) Python 3 builtins. Provides functions that exist in 2.x, but have different behavior in Python 3 (ascii, map, filter, hex...). To write Python 3 compatible code, import the fcts from this module, e.g.:
	`from future_builtins import map`
	`...code using Python3-syle map()...`

gc	Perform garbage collection, obtain GC debug stats, and tune GC parameters. 2.5: New `get_count()` function. `gc.collect()` takes a new *generation* argument.
gdbm	GNU's reinterpretation of dbm [Unix].
getopt	Standard command line processing. See also optparse.

getpass	Utilities to get a password and/or the current user name.
gettext	Internationalization and localization support.
glob	Filename "globbing" utility.
~~gopherlib~~	~~Gopher protocol client interface.~~
grp	The group database [Unix].
~~grep~~	~~'grep' utilities.~~
gzip	Read & write gzipped files.
hashlib	Secure hashes and message digests.
heapq	Heap queue (priority queue) helpers. 2.5: `nsmallest()` and `nlargest()` takes a `key` keyword param.
hmac	HMAC (Keyed-Hashing for Message Authentication).
hotshot.stones	Helper to run the pystone benchmark under the Hotshot profiler.
htmlentitydefs	HTML character entity references.
htmllib	HTML2 parsing utilities
HTMLParser	Simple HTML and XHTML parser.
httplib	HTTP1 client class.
idlelib	(package) Support library for the IDLE development environment.
ihooks	Hooks into the "import" mechanism. Deprecated since 2.6.
imageop	Manipulate raw image data. Deprecated since 2.6.
imaplib	IMAP4 client.Based on RFC 2060.
imghdr	Recognizing image files based on their first few bytes.
imp	Access the import internals.
imputil	Provides a way of writing customized import hooks.
inspect	Get information about live Python objects.
io	(new in 2.6) Core tools for working with streams [PEP 3116]. Define Abstract Base Classes `RawIOBase` (I/O operations: read, write, seek..), `BufferedIOBase` (buffering), and `TextIOBase` (reading & writing strings).
itertools	Tools to work with iterators and lazy sequences. 2.5: `islice()` accepts `None` for start & step args. 2.6: Several new functions: `izip_longest`, `product`, `combinations`, `permutations`.
json	(new in 2.6) JSON (JavaScript Object Notation) interchange format support.
keyword	List of Python keywords.
~~knee~~	~~A Python re-implementation of hierarchical module import.~~
linecache	Cache lines from files.
~~linuxaudiodev~~	~~Linux /dev/audio support.~~ Replaced by `ossaudiodev`(Linux).
locale	Support for number formatting using the current locale settings. 2.5: `format()` modified; new fcts `format_string()` and `currency()`
logging	(package) Tools for structured logging in log4j style.
macpath	Pathname (or related) operations for the Macintosh [Mac].
macurl2path	Mac specific module for conversion between pathnames and URLs [Mac].
mailbox	Classes to handle Unix style, MMDF style, and MH style mailboxes. 2.5: added capability to modify mailboxes in addition to reading them.
mailcap	Mailcap file handling (RFC 1524).
marshal	Internal Python object serialization.
markupbase	Shared support for scanning document type declarations in HTML and XHTML.
math	Mathematical functions. See also cmath
md5	MD5 message digest algorithm. 2.5: Now a mere wrapper around new library `hashlib`. **Deprecated** since 2.6, use `hashlib` module instead.
mhlib	MH (mailbox) interface. Deprecated since 2.6.
mimetools	Various tools used by MIME-reading or MIME-writing programs. Deprecated since 2.6.
mimetypes	Guess the MIME type of a file.
MimeWriter	Generic MIME writer. **Deprecated** since 2.3, use `email` package instead.
mimify	Mimification and unmimification of mail messages. **Deprecated** since 2.6, use `email` package instead.
mmap	Interface to memory-mapped files - they behave like mutable strings.
modulefinder	Tools to find what modules a given Python program uses, without actually running the program.
msilib	Read and write Microsoft Installer files [Windows].
msvcrt	File & Console Windows-specific operations [Windows].
multifile	A readline()-style interface to the parts of a multipart message. Deprecated since 2.6.

multiprocessing	(new in 2.6) Process-based "threading" interface. Allows to fully leverage **multiple processors** on a machine [Windows, Unix] [PEP 371].
mutex	Mutual exclusion -- for use with module sched. See also std module `threading`, and glock.
netrc	Parses and encapsulates the netrc file format.
new	Creation of runtime internal objects (interface to interpreter object creation functions). Deprecated since 2.6.
nis	Interface to Sun's NIS (Yellow Pages) [Unix]. 2.5: New *domain* arg to `nis.match()` and `nis.maps()`.
nntplib	An NNTP client class. Based on RFC 977.
ntpath	Common operations on Windows pathnames [Windows].
nturl2path	Convert a NT pathname to a file URL and vice versa [Windows].
numbers	(new in 2.6) **Numeric Abstract Base Classes (ABC)** [PEP 3141]. Define a type hierarchy for numbers: `Number`, `Complex`, `Real`, `Rational`, `Integral`.
olddifflib	Old version of difflib (helpers for computing deltas between objects)?
operator	Standard operators as functions. 2.5: `itemgetter()` and `attrgetter()` now supports multiple fields.
optparse	Improved command-line option parsing library (see also getopt). 2.5: Updated to Optik library 1.51.
os	OS routines for Mac, DOS, NT, or Posix depending on what system we're on. 2.5: `os.stat()` return time values as floats; new constants to `os.lseek()`; new functions `wait3()` and `wait4()`; on FreeBSD, `os.stat()` returns times with nanosecond resolution.
os.path	Common pathname manipulations.
os2emxpath	os.path support for OS/2 EMX.
~~packmail~~	~~Create a self-unpacking shell archive.~~
parser	Access Python parse trees.
pdb	A Python debugger.
pickle	Pickling (save/serialize and restore/deserialize) of Python objects (a faster C implementation exists in built-in module: `cPickle`). 2.5: Value returned by `__reduce__()` must be different from `None`.
pickletools	Tools to analyze and disassemble pickles.
pipes	Conversion pipeline templates [Unix].
pkgutil	Tools to extend the module search path for a given package. 2.5: PEP302's import hooks support; works for packages in ZIP format archives.
platform	Get info about the underlying platform.
~~poly~~	~~Polynomials.~~
popen2	Spawn a command with pipes to its stdin, stdout, and optionally stderr. Superseded by module subprocess since 2.4. Deprecated since 2.6.
poplib	A POP3 client class.
posix	Most common POSIX system calls [Unix].
posixfile	**(deprecated since 1.5, use `fcntl.lockf()` instead)** File-like objects with locking support [Unix].
posixpath	Common operations on POSIX pathnames.
pprint	Support to pretty-print lists, tuples, & dictionaries recursively.
pre	Support for regular expressions (RE) - see `re`.
profile	Class for profiling python code. 2.5: See also new fast C implementation `cProfile`
pstats	Class for printing reports on profiled python code. 2.5: new *stream* arg to `Stats` constructor.
pty	Pseudo terminal utilities [Linux, IRIX].
pwd	The password database [Unix].
py_compile	Routine to "compile" a .py file to a .pyc file.
pyclbr	Parse a Python file and retrieve classes and methods.
pydoc	Generate Python documentation in HTML or text for interactive use.
pyexpat	Interface to the Expat XML parser. 2.5: now uses V2.0 of the expat parser.
~~PyUnit~~	~~Unit test framework inspired by JUnit.~~ See `unittest`.
Queue	A multi-producer, multi-consumer queue. 2.6: New queue variants `PriorityQueue` and `LifoQueue`.
quopri	Conversions to/from quoted-printable transport encoding as per RFC 1521.
rand	Don't use unless you want compatibility with C's rand().
random	Random variable generators.

re	Regular Expressions.
readline	GNU readline interface [Unix].
~~reconvert~~	~~Convert old ("regex") regular expressions to new syntax ("re").~~
~~regexp~~	~~Backward compatibility for module "regexp" using "regex".~~
~~regex_syntax~~	~~Flags for `regex.set_syntax()`.~~
~~regsub~~	~~Regexp-based split and replace using the obsolete regex module.~~
repr	Alternate `repr()` implementation.
resource	Resource usage information [Unix].
~~rexec~~	~~Restricted execution facilities ("safe" exec, eval, etc).~~
rfc822	Parse RFC-8222 mail headers.
~~rgbimg~~	~~Read and write 'SGI RGB' files.~~
rlcompleter	Word completion for GNU readline 2.0 [Unix]. 2.5: Doesn't depend on `readline` anymore; **now works on non-Unix platforms**.
robotparser	Parse robot.txt files, useful for web spiders.
sched	A generally useful event scheduler class.
select	Waiting for I/O completion.
sets	A Set datatype implementation based on dictionaries. Deprecated since 2.6, use built-in types `set` and `frozenset` instead.
sgmllib	A parser for SGML, using the derived class as a static DTD.
sha	SHA-1 message digest algorithm. 2.5: Now a mere wrapper around new library `hashlib`. Deprecated since 2.6, use `hashlib` instead.
shelve	Manage shelves of pickled objects.
shlex	Lexical analyzer class for simple shell-like syntaxes.
shutil	Utility functions for copying files and directory trees.
signal	Set handlers for asynchronous events.
SimpleHTTPServer	Simple HTTP Server.
SimpleXMLRPCServer	Simple XML-RPC Server. 2.5: New attribute `rpc_paths`.
site	Append module search paths for third-party packages to `sys.path`.
smtpd	An RFC 2821 SMTP server.
smtplib	SMTP/ESMTP client class.
sndhdr	Several routines that help recognizing sound.
socket	Socket operations and some related functions. Now supports timeouts thru function `settimeout(t)`. Also supports SSL on Windows. 2.5: Now supports `AF_NETLINK` sockets on Linux; new socket methods `recv_buf(`*buffer*`)`, `recvfrom_buf(`*buffer*`)`, `getfamily()`, `gettype()` and `getproto()` .
SocketServer	Generic socket server classes.
spwd	Access to the UNIX shadow password database [Unix].
sqlite3	DB-API 2.0 interface for SQLite databases.
sre	Support for regular expressions (RE). See re.
stat	Constants/functions for interpreting results of os.
statcache	Maintain a cache of `stat()` information on files.
statvfs	Constants for interpreting statvfs struct as returned by `os.statvfs()` and `os.fstatvfs()` (if they exist). Deprecated since 2.6.
string	A collection of string operations (see Strings).
StringIO	File-like objects that read/write a string buffer (a faster C implementation exists in built-in module cStringIO).
stringprep	Normalization and manipulation of Unicode strings.
struct	Perform conversions between Python values and C structs represented as Python strings. 2.5: faster (new `pack()` and `unpack()` methods); pack and unpack to and from buffer objects via methods `pack_into` and `unpack_from`.
subprocess	Subprocess management. Replacement for os.system, os.spawn*, os.popen*, popen2.* [PEP324]
sunau	Stuff to parse Sun and NeXT audio files.
sunaudio	Interpret sun audio headers.
symbol	Non-terminal symbols of Python grammar (from "graminit.h").
symtable	Interface to the compiler's internal symbol tables.
sys	System-specific parameters and functions.
syslog	Unix syslog library routines [Unix].
tabnanny	Check Python source for ambiguous indentation.
tarfile	Tools to read and create TAR archives. 2.5: New method `TarFile.extractall()`.
telnetlib	TELNET client class. Based on RFC 854.
tempfile	Temporary files and filenames. 2.6: New classes `SpooledTemporaryFile` and

	NamedTemporaryFile.
termios	POSIX style tty control [Unix].
test	Regression tests package for Python.
textwrap	Tools to wrap paragraphs of text.
thread	Multiple threads of control (see also threading below).
threading	New threading module, emulating a subset of Java's threading model. 2.5: New function stack_size([*size*]) allows to get/set the stack size for threads created. 2.6: Several functions renamed or replaced by properties, new property Thread.ident. See also new module multiprocessing.
threading_api	(doc of the threading module).
time	Time access and conversions.
timeit	Benchmark tool.
Tix	Extension widgets for Tk.
Tkinter	Python interface to Tcl/Tk.
toaiff	Convert "arbitrary" sound files to AIFF (Apple and SGI's audio format). Deprecated since 2.6.
token	Token constants (from "token.h").
tokenize	Tokenizer for Python source.
trace	Tools to trace execution of a function or program.
traceback	Extract, format and print information about Python stack traces.
tty	Terminal utilities [Unix].
turtle	LogoMation-like turtle graphics.
types	Define names for all type symbols in the std interpreter.
tzparse	Parse a timezone specification.
unicodedata	Interface to unicode properties. 2.5: Updated to Unicode DB 4.1.0; Version 3.2.0 still available as unicodedata.ucd_3_2_0. 2.6: Updated to Unicode DB 5.1.0.
unittest	Python unit testing framework, based on Erich Gamma's and Kent Beck's JUnit.
urllib	Open an arbitrary URL.
urllib2	An extensible library for opening URLs using a variety of protocols.
urlparse	Parse (absolute and relative) URLs.
user	Hook to allow user-specified customization code to run.
~~UserDict~~	A wrapper to allow subclassing of built-in dict class (useless with *new-style* classes. Since Python 2.2, dict is subclassable).
~~UserList~~	A wrapper to allow subclassing of built-in list class (useless with *new-style* classes. Since Python 2.2, list is subclassable)
~~UserString~~	A wrapper to allow subclassing of built-in string class (useless with *new-style* classes. Since Python 2.2, str is subclassable).
~~util~~	~~some useful functions that don't fit elsewhere !!~~
uu	Implementation of the UUencode and UUdecode functions.
uuid	UUID objects according to RFC 4122.
warnings	Python part of the warnings subsystem. Issue warnings, and filter unwanted warnings.
wave	Stuff to parse WAVE files.
weakref	Weak reference support for Python. Also allows the creation of proxy objects. 2.5: new methods iterkeyrefs(), keyrefs(), itervaluerefs() and valuerefs().
webbrowser	Platform independent URL launcher. 2.5: several enhancements (more browsers supported, etc...).
~~whatsound~~	~~Several routines that help recognizing sound files.~~
whichdb	Guess which db package to use to open a db file.
whrandom	Wichmann-Hill random number generator (obsolete, use random instead).
winsound	Sound-playing interface for Windows [Windows].
wsgiref	WSGI Utilities and Reference Implementation.
xdrlib	Implements (a subset of) Sun XDR (eXternal Data Representation).
xmllib	A parser for XML, using the derived class as static DTD.
xml.dom	Classes for processing XML using the DOM (Document Object Model). 2.3: New modules expatbuilder, minicompat, NodeFilter, xmlbuilder.
xml.etree.ElementTree	Subset of Fredrik Lundh's ElementTree library for processing XML.
xml.parsers.expat	An interface to the Expat non-validating XML parser.
xml.sax	Classes for processing XML using the SAX API.
xmlrpclib	An XML-RPC client interface for Python. 2.5: Supports returning datetime objects for the XML-RPC date type.
xreadlines	Provides a sequence-like object for reading a file line-by-line without reading the

	entire file into memory. Deprecated since release 2.3. Use `for line in file` instead. Removed since 2.4
zipfile	Read & write PK zipped files. 2.5: Supports ZIP64 version, a .zip archive can now be larger than 4GB. 2.6: Class `ZipFile` has new methods `extract()` and `extractall()`.
zipimport	ZIP archive importer.
zlib	Compression compatible with gzip. 2.5: `Compress` and `Decompress` objects now support a `copy()` method.
~~zmod~~	~~Demonstration of abstruse mathematical concepts.~~

Workspace exploration and idiom hints

`dir(`*`object`*`)`	list valid attributes of *object* (which can be a module, type or class object)
`dir()`	list names in current local symbol table.
`if __name__ == '__main__':` `    main()`	invoke main() if running as script
`map(None, lst1, lst2, ...)`	merge lists; see also zip(lst1, lst2, ...)
`b = a[:]`	create a copy b of sequence a
`b = list(a)`	If a is a list, create a copy of it.
`a,b,c = 1,2,3`	Multiple assignment, same as a=1; b=2; c=3
`for key, value in dic.items():` `...`	Works also in this context
`if 1 < x <= 5: ...`	Works as expected
`for line in fileinput.input():` `...`	Process each file in command line args, one line at a time
`_`	(underscore) in interactive mode, refers to the last value printed.

Python Mode for Emacs

Emacs goodies available here.

(The following has not been revised, probably not up to date - any contribution welcome -)

```
Type C-c ? when in python-mode for extensive help.
INDENTATION
Primarily for entering new code:
        TAB      indent line appropriately
        LFD      insert newline, then indent
        DEL      reduce indentation, or delete single character
Primarily for reindenting existing code:
        C-c :    guess py-indent-offset from file content; change locally
        C-u C-c :        ditto, but change globally
        C-c TAB  reindent region to match its context
        C-c <    shift region left by py-indent-offset
        C-c >    shift region right by py-indent-offset
MARKING & MANIPULATING REGIONS OF CODE
C-c C-b         mark block of lines
M-C-h           mark smallest enclosing def
C-u M-C-h       mark smallest enclosing class
C-c #           comment out region of code
C-u C-c #       uncomment region of code
MOVING POINT
C-c C-p         move to statement preceding point
C-c C-n         move to statement following point
C-c C-u         move up to start of current block
M-C-a           move to start of def
C-u M-C-a       move to start of class
M-C-e           move to end of def
C-u M-C-e       move to end of class
EXECUTING PYTHON CODE
C-c C-c sends the entire buffer to the Python interpreter
C-c |   sends the current region
C-c !   starts a Python interpreter window; this will be used by
        subsequent C-c C-c or C-c | commands
VARIABLES
```

```
py-indent-offset        indentation increment
py-block-comment-prefix comment string used by py-comment-region
py-python-command       shell command to invoke Python interpreter
py-scroll-process-buffer        t means always scroll Python process buffer
py-temp-directory       directory used for temp files (if needed)
py-beep-if-tab-change   ring the bell if tab-width is changed
```